Satellite Relative Motion Control for MIT's SPHERES Program

THESIS

Samuel P. Barbaro, Second Lieutenant, USAF

AFIT/GA/ENY/12-M02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GA/ENY/12-M02

Satellite Relative Motion Control for MIT's SPHERES Program

THESIS

Presented to the Faculty

Department of Aeronautics and Astronautics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Astronautical Engineering

Samuel P. Barbaro, B.S.

Second Lieutenant, USAF

March 2012

AFIT/GA/ENY/12-M02

Satellite Relative Motion Control for MIT's SPHERES Program

Samuel P. Barbaro, B.S.

Second Lieutenant, USAF

Approved:

/signed/	7 Mar 2012
Dr. Richard G. Cobb (Chairman)	date
/signed/	7 Mar 2012
Col. Timothy L. Lawrence, PhD (Member)	date
/signed/	7 Mar 2012
LtCol Ronald J. Simmons, PhD (Member)	date

Abstract

Autonomous formation flight concepts and algorithms have great potential to revolutionize spacecraft operations enabling missions to perform autonomous docking, in-space refueling, in-space robotic assembly, and space debris removal. Such tasks require the implementation of speed and path control algorithms to maneuver satellites along relative paths with specified rates along those paths. This thesis uses MATLAB® and SIMULINK® to design and simulate a control algorithm capable of providing relative speed and path control between satellites with a pointing error of less than two degrees, a position error of less than two millimeters, and a millimeter per second of velocity error. The enclosed research provides enhancements to Massachusetts Institute of Technology's SPHERES (Synchronized Position Hold Engage Reorient Experimental Satellites) program, a testbed for multi-object rendezvous and docking research. This control algorithm is to be used on-board the International Space Station to allow MIT's SPHERES program to continue to provide a practical intermediate step to develop, test, and validate autonomous formation spaceflight algorithms. Furthermore, the simulation tool used to develop the control algorithm allows a greater community of control engineers to interact with SPHERES purely in the MATLAB® development environment.

Acknowledgements

I cannot begin to adequately convey the gratitude owed to the many individuals responsible for helping me complete this research. Let me first give thanks to the One who makes all things possible—it is through Him I derive my strength and for Him I labor.

The faculty and staff at AFIT have helped me every step of the way, and I will always be grateful. I would especially like to thank Dr. Swenson and Dr. Black for letting me bounce ideas off of them, and for helping me better understand how to work with quaternions.

I would like to thank the sponsor of my research, Dr. David Miller with the Space Systems Laboratory of MIT for giving me the opportunity to work with the SPHERES program. I would also like to thank my MIT point-of-contact, 2Lt Micheal O'Connor who was a great help in proving me with the necessary information on the SPHERES program.

I could never have been able to complete the program here without my wonderful wife who pushed me and supported me the whole way. Of course, my classmates supported me every step of the way as well, and I would especially like to thanks 2Lt Rob Steigerwald for his insight and knowledge of just about everything.

Lastly, I owe a sincere debt of gratitude to my advisor, Dr. Rich Cobb. He spent countless hours discussing the project with me, pulling me through my problems with MATLAB®, and he helped shape the research into its final state. Thank you.

Samuel P. Barbaro

Table of Contents

Page

List of Figures

List of Tables

List of Symbols

Symbol		Page
$\bar{q}_{error}$	Quaternion Error	50
τ	'Look Ahead' Gain	52
Q_1	LQR Position Weight	56
Q_2	LQR Velocity Weight	56
R_1	LQR Control Weight	56
δ	Dead-Zone Limit	79
$\bar{x}$	Vector	87
$\bar{m}$	Slope Vector	87
$\bar{b}$	Intercept Vector	87
$\tilde{r}_t$	User-Supplied Location of Target	89
$\tilde{\rho}$	Desired Range Inspector is from Target	89
$\bar{r}_t$	True Location of Target	89
$\bar{e}_t$	Error Between Desired and True Location of Target	89
$\tilde{r}_{in}$	Desired Location of Inspector	91
$\bar{\rho}$	Actual Range Inspector is from Target	92

List of Abbreviations

Satellite Relative Motion Control for MIT's SPHERES Program

I. Introduction

Spacecraft formation-flying techniques and satellite autonomy can transform the way space missions are conducted because these concepts introduce innovative mission capabilities to the domain of space. Specifically, autonomous formation spaceflight techniques with the use of speed and path control provide users with a capability to routinely maneuver to provide autonomous satellite docking procedures and in-space refueling.

The United States Air Force (USAF) affirms that space capabilities are a vital aspect of air and space power [6]. Spacecraft with speed and path control algorithms provide an unprecedented level of flexibility to the operational functions of the Air Force in space through the means of autonomous docking, in-space robotic assembly, debris removal, and in-space refueling. The manner in which autonomous formation spaceflight with speed and path control algorithms can accomplish this is discussed in Sections 1.1 and 1.2.

The technology for autonomous formation spaceflight is still an emerging concept that needs further development. Before satellites can be used in this manner, the spacecraft will need to have algorithms capable of controlling these spacecraft relative to other objects in space. A number of institutions within the scientific community including Massachusetts Institute of Technology (MIT) are conducting research to further the development of autonomous formation spaceflight. Working cooperatively with MIT, this thesis is focused on providing MIT with a speed and path control algorithm to be integrated with their work to demonstrate precise control of satellites operating relative to other satellites. Section 1.3 discusses how MIT is working to continue to develop formation spaceflight, and Section 1.4 introduces the research

within this thesis and how this thesis will be used to contribute to the understanding of autonomous formation spaceflight.

1.1 Autonomous Docking

The ability for spacecraft to autonomously dock with space objects would definitely find use within the USAF. Air Force doctrine specifies the need for the Air Force to sustain existing space systems, augment these systems with redundant or additional capabilities as national needs dictate, and service or maintain these space systems [6]. Space system sustainment is required for space systems whose individual satellites need to be replaced because it has failed or is predicted to fail. These needs however are currently attained through the costly process of space lift. Autonomous docking provides for another means of sustainment, augmentation, and maintenance of space systems.

The vast majority of satellites launched to date have not been resupplied, serviced, upgraded, or reconfigured while on orbit. This basic operational limitation could be changed by developing robust autonomous docking control algorithms and the associated servicing equipment. Autonomous algorithms would eliminate the need for complicated maneuvers executed by large and expensive ground operation teams.

Autonomous rendezvous and docking could be used to restore mission capability to satellites that are tumbling or spinning uncontrollably. Routines could also be developed to allow one satellite (or a number of smaller satellites) to approach, dock, determine new mass moments of inertia of the combined system, and thrust in a manner to restore a desired orientation and stabilize the satellite before releasing. This would allow the now stable satellite to use its own control systems to resume normal stability procedures for the spacecraft. This could in effect save multi-million dollar programs. A specific example is the Astra 5A commercial telecommunications satellite launched in 1997 [7]. In January of 2009, that satellite lost control of its orientation after experiencing a technical anomaly [7]. The satellite was then unable to charge its batteries with its solar panels and then ceased functioning. Thus without

power the satellite became useless which resulted in a loss of millions of dollars [7]. Autonomous spaceflight technologies with the capability to autonomously dock could have prevented this loss by stabilizing Astra 5A, and orienting the satellite's solar panels to the sun to recharge the batteries. At this point the ground crews on Earth could diagnose errors and potentially restore Astra 5A back to full mission readiness.

Furthermore, this process could also be used to propel misplaced satellites that did not make it to their desired mission orbits. An example of such a satellite would be the Air Force's billion dollar AEHF-1 satellite [8] that failed to reach its desired orbit when an apogee motor failed to ignite[1]. Autonomous docking could also be used for in-space robotic assembly and the removal of space debris.

1.1.1 In-Space Robotic Assembly. Speed and path control algorithms can also be used to provide methods to explore new space capabilities through use of autonomous robotic space assembly. This concept involves using satellites as robotic workers that could be programmed to build various structures in space. Instead of using astronauts to assemble the structures in space, small satellites acting as robots could do the work continuously only stopping to refuel at nearby mother ships or to wait for more materials to be launched. Robust control algorithms would allow for these small satellites to operate safely in the harsh environment without endangering human life [10]. This ability would potentially allow for large structures to be created in space faster, cheaper, and safer than current methods allow. Although MIT is mostly interested in this concept, the USAF would likely make use of this concept to develop for future missions that require space structures that are much larger than what can currently launch atop a single booster.

1.1.2 Space Debris Removal. Speed and path control algorithms in conjunction with autonomous formation spaceflight techniques can also be used to provide kinetic operations to attain and maintain space superiority through the removal of

[1]While the apogee motor failed, the program was saved by using on board Hall-effect thrusters to eventually boost AEHF-1 to the proper orbit after nine months [9].

space debris. Space debris includes any man-made object orbiting the Earth that no longer has a useful purpose [11]. There are currently over 19,000 known objects of space debris greater than ten centimeters. This includes left over upper stage rockets, defunct satellites, and debris from spacecraft collisions [12]. This space junk poses an increasing risk to space capabilities as the amount of debris continues to grow. The risk of debris impacting space capabilities is epitomized by the 2009 collision of the inoperative Cosmos 2251 with the Iridium 33 communications satellite [13]. This collision not only affected Iridium Communications Inc.®, but the collision also affects all space users because the collision added over one thousand pieces of debris larger than ten centimeters [14] which increases the chances of future collisions, especially since methods to remove space debris do not currently exist.

Spacecraft speed and path control can be used to remedy this growing problem, and directly integrates with USAF space doctrine for the purpose of debris removal. Among a few of the tenets of defensive counter space, spacecraft speed and path control allows the USAF to pro-actively preserve space capabilities, restore and recover space capabilities, and suppress threats to friendly space capabilities [6]. Speed and path control algorithms could be used on a number of small satellites operating from a larger spacecraft to allow the small satellites to attach themselves to large pieces of space debris. These satellites could then force the debris to re-enter the Earth's atmosphere on a trajectory that would allow the space junk to burn up on descent after the small satellites detached and returned to the larger spacecraft. When used in conjunction with in-space refueling (Section 1.2), these satellites could be used to remove a number of large space debris throughout their mission lifetime. This method would allow the USAF to protect friendly space capabilities from the threats posed by space debris.

1.2 In-Space Refueling

Like aircraft, spacecraft are limited by the amount of available fuel. The amount of fuel carried by a spacecraft plays a part in determining the mission length, payload

mass, and the reliability the spacecraft will have throughout the course of its mission. In addition, situations exist in which it would be desirable for autonomous satellites to temporarily disregard the need to fly in Keplarian orbits. Previously, this desire has been ignored because taking satellites out of Keplarian orbits typically requires fuel to be consumed at an unacceptable rate. Thus, without the use of in-space refueling, spacecraft would not be able to violate Keplarian orbits for any useful length of time. However, autonomous formation spaceflight with the use of speed and path control algorithms provide the framework to make in-space refueling a reality. This concept not only allows for satellites to temporarily leave Keplarian orbits as missions dictate, but it would also allow for other spacecraft to carry less fuel, and operate much longer than current satellites can. Refueling satellites in space can extend service life, reduce launch costs by reducing fuel mass of satellites, and provide an opportunity for satellites to occasionally ignore Keplarian orbits and be used in numerous applications which current satellites are unable to perform. Space refueling allows for satellites to consist of heavier payloads without having to sacrifice payload mass for fuel mass. This in turn makes space refueling a force multiplier much like aerial refueling provides greater capabilities for missions within the Earth's atmosphere [6].

A space-based laser system is one way the USAF could take advantage of in space refueling. The concept for a space-based laser was popularized by President Reagan in 1983 with his proposal of the Strategic Space Initiative [15]. This plan sought to use a space-based laser for ballistic missile defense. Space lasers have also been considered for removal of orbital debris between one and ten centimeters [16]. Regardless of the use for a space-based laser, one major drawback is that chemical lasers could only be fired a few times before the fuel for the laser was expended [17]. This would traditionally mean that the satellite with the laser could no longer serve its intended purpose. Formation spaceflight, through the use of speed and path control algorithms, could change that by implementing space refueling. Another satellite(s) could be used to deliver the fuel that powers the laser which would considerably extend the usable lifetime of a space-based laser system.

1.3 MIT's SPHERES Testbed

As introduced in the preceding examples, speed and path controllers, have the potential to provide new and innovative ways to improve methods for conducting space operations using autonomous formation spaceflight. Yet given the high cost of operating spacecraft from the ground where in-contact times are often only a small fraction of an orbit period, there is a strong incentive to perform the tight control of relative position and orientation autonomously. But as with any emerging technology, a high degree of risk is inherently applied when creating and applying formation flight and docking control algorithms to real-world space systems. As these programs are oftentimes multi-million or multi-billion dollar systems, the risk becomes intolerable. In order to reduce risks associated with autonomous formation spaceflight, and to a greater extent formation spaceflight technologies, MIT has developed the SPHERES (Synchronized Position Hold Engage Re-orient Experimental Satellites) testbed as a practical intermediate step to develop, test, and validate autonomous algorithms.

SPHERES is a spacecraft formation flying testbed designed to provide a cost-effective, long duration, re-loadable, and easily reconfigurable platform with representative dynamics for the development and validation of metrology, formation flying, and autonomy algorithms [2]. Their algorithms are intended to help the Air Force and NASA buy down the high risk associated with autonomous rendezvous and docking algorithms. Figure 1.1 shows what the small satellite looks like.

The SPHERES testbed currently has two test locations: MIT's Space System Laboratory and the International Space Station (ISS). The Space System Laboratory is located at MIT and provides users with a two dimensional 1-g test environment, while the ISS provides users with an environment to exploit the effects of micro gravity and test SPHERES in all three dimensions.

Figure 1.1: SPHERES Satellite [1]

1.4 Thesis Problem Statement

To further the development of formation-flying technologies, this thesis will investigate enhancements to the SPHERES software control suite operating under MIT's Guest Scientist Program [5]. Currently, SPHERES uses control algorithms that close the loop on the satellite's position in the body frame and its orientation in the global frame. This allows users to dictate how one SPHERES should be positioned and orientated with respect to another SPHERES satellite. Although these algorithms have led to many successful SPHERES tests to date, the SPHERES platform is currently unable to maneuver along a path while simultaneously controlling the velocity of the satellite. This thesis aims to produce a control algorithm to remedy this deficiency.

Having introduced the motivation and objective for the current research, the thesis work is documented as follows. Chapter II provides a review of coordinate frames, quaternions, control strategies and related research in three dimensional trajectory tracking, as well as a background on how SPHERES has been used in previous research. Next, Chapter III develops the methodology applied to the design of the speed and path controller for SPHERES and provides initial simulated results. Fol-

lowing this, Chapter IV documents the results from testing and discusses how to best implement the control algorithm with MIT's SPHERES program. Lastly, Chapter V offers conclusions from the speed and path control algorithm designed herein.

II. Background

The study of relative motion between spacecraft is not a new concept. This background provides a brief synopsis of the past work within the field of formation spaceflight. Additionally, reference frames, their rotations, and a number of control techniques are included to understand how to develop a control algorithm for the relative motion control of a satellite. Section 2.1 introduces some of the previous research in formation spaceflight and provides a few historical examples. Section 2.2 defines the coordinate frames used through this thesis. Section 2.3 presents methods for rotating between these coordinate frames. Section 2.4 covers the basic control techniques that are used to develop the control algorithm with this research. Lastly, Section 2.5 examines the dynamics used to govern the satellites used in the designed simulation and demonstrates how the SPHERES plant is modeled.

2.1 Formation Spaceflight Throughout History

From 1983 to 2005, fifty-seven shuttle missions successfully utilized one or more forms of close proximity operations. But formation spaceflight did not begin solely with the Shuttle Program. Experiments to validate the ability of a human eye to track and maintain control of a docking sequence were preformed on Mercury missions. Following the Mercury Program, NASA's Gemini program sought to improve and provide a firm foundation for manual rendezvous and docking procedures [18]. During Project Gemini, rendezvous and docking technology and mission techniques were developed and successfully demonstrated. Additionally, Goodman states that the most significant accomplishments of the Gemini program with respect to rendezvous operations included multiple rendezvous operations while staying within a propellant budget [19]. Next the Apollo Program capitalized on the research of the Gemini Program and included rendezvous operations as methodical techniques, using several missions to practice lunar landing. By the time the Shuttle Program contracts were awarded in 1972, rendezvous, docking technology, and flight techniques were considered to be mature and the challenges well understood. This allowed more

automation[1] to be included in the design of the Shuttle rendezvous procedures used during the construction of the ISS [18].

The rendezvous and docking procedures developed for NASA were a result of W.H. Clohessy and R.S. Wiltshire's development of relative equations of motion in the early 1960's [20]. The Clohessy-Wiltshire equations not only allow for docking and rendezvous procedures, but also close-proximity operations between spacecraft. Although the Clohessy-Wiltshire equations only account for the main satellite or 'chief' to have a circular orbit, this restriction can be removed through more complex sets of these equations which have been developed in recent years [21].

Research into formation spaceflight is a topic of growing interest as the potential advantages to be gained through coordinated satellite formations are brought to light. This can be seen by observing the number of government programs dedicated to formation spaceflight and its related technological development. The TechSat-21 program investigated emerging technologies essential for satellite formations [22]. Although TechSat-21 was canceled in 2003, the program was meant to be a technology demonstrator for distributed mission architecture, micro-satellite bus, micro-propulsion, sparse aperture sensing, and collaborative behavior. The Air Force Research Laboratory (AFRL), in an effort to establish proximity operations with small satellites, launched the XSS-11 on 11 April, 2005. This satellite successfully demonstrated rendezvous and proximity operations with an expended rocket body as well as several US-owned inactive space objects near its orbit [23]. NASA also launched the Demonstration of Autonomous Rendezvous Technology (DART) program in April 2005. This program was part of NASA's efforts to make space travel safer and more affordable by demonstrating technologies for spacecraft to autonomously locate and rendezvous with other spacecraft without direct human guidance [24]. This program successfully demonstrated the capability to locate and rendezvous, but was unable to perform all of the close-proximity and circumnavigation tasks when it ran out of fuel.

[1] Although these techniques were designed with more automation, astronauts still were involved with all formation procedures.

The Defense Advanced Research Projects Agency (DARPA) has also sought to validate a variety of proximity operations. Specifically, DARPA's Orbital Express Space Operations Architecture demonstrated the ability for autonomous on-orbit refueling and reconfiguration of two satellites [25]. This program successfully launched on 8 March 2007 and completed the technology demonstration on 22 July 2007.

Interest in formation spaceflight is not limited to the government industry. MIT has pursued research in this field through the use of SPHERES. The SPHERES program provides a testbed with six degrees-of-freedom on-board the ISS [5]. The SPHERES testbed has demonstrated the capability for two satellites to create, maintain or leave a formation. A third satellite has also been shown to be capable of joining an existing formation. Path planning algorithms have also been installed to provide the ability for one SPHERES to dock with another uncooperative spacecraft that is freely tumbling [26]. The SPHERES program was first brought to the ISS in 2003 but research is still on-going. Future plans for SPHERES include the installation of computer vision based navigation [27]. Merging this capability with a speed and path control algorithm would provide for a number of new concepts to be explored.

2.2 Reference Frames

Before transition into the design of the satellite controller, it is fundamental to understand the satellite's dynamics as well as how those dynamics change with the frame of reference. In addition, a number of reference frames are used both within this thesis and when working in space in general because certain frames of reference provide specific advantages. Section 2.2.1 discusses the use of an inertial reference frame and Section 2.2.2 details what the body frame is and how the frame is used. It is worth noting that in the scope of this thesis all reference frames consist of a right-hand coordinate frame of three orthogonal unit vectors.

2.2.1 Inertial Reference Frame. An inertial coordinate system uses a frame of reference that does not accelerate and has constant rectilinear motion with respect

to any other inertial reference frame. Furthermore, Newton's laws must be expressed in an inertial reference frame [28]. To illustrate this, consider Newton's Second Law of motion, Equation 2.1. Let 'm' represent the mass of the object, and let 'A' be defined as a vector consisting of the object's acceleration as viewed from an inertial frame of reference.

$$F = mA \tag{2.1}$$

Equation 2.1 produces a force vector that consists of all the true forces (ie. gravitational, electro-magnetic, nuclear, etc.) acting on the object. Although this fundamental equation is a true representation of force, it is only valid in an inertial reference frame that does not accelerate in relation to the observed object [28]. If the frame of reference is also accelerating, Newton's Second Law must be expanded to include the additional accelerations of the frame of reference as shown in Equation 2.2 [29]. Let 'A' denote an acceleration vector and 'V' refer to a velocity vector. Next, the subscript 'obj' refers to the object in question, and 'rf' denotes a value of the reference frame. Furthermore, the variable $R_{obj/rf}$, identifies the position of the object in the frame of reference and ω is the angular velocity of the reference frame.

$$F = m(A_{obj} + A_{rf} + 2\omega \times V_{rf} + \alpha \times R_{obj/rf} + \omega \times (\omega \times R_{obj/rf})) \tag{2.2}$$

Equation 2.2 now defines a more complicated force vector that represents the force of the object as seen from the moving reference frame. This force results from not only the acceleration of the object and the acceleration of the moving reference frame with respect to inertial space (as seen in the first two terms of Equation 2.2), but the force also is affected by a Coriolis acceleration, centrifugal acceleration, and an Euler acceleration [29]. These additional accelerations result from the relative motion between the object and the moving reference frame and can be thought to include

'fictitious' forces on the object. The reason these forces are considered fictitious is because they do not affect the object as viewed from inertial space because the forces do not actually exist.

To clarify this concept, consider a simple game of catch. If a person is stationary and tossed a ball, they could intuitively use Newton's Second Law to catch the ball with ease. But if they were to play that same game of catch while skydiving things would be more complex. If tossed a ball while tumbling through the air, the act of catching the ball becomes considerably more difficult. This is because the ball would appear to move differently from the catcher's perspective. The ball is not moving differently than it did before. It's the catcher that does the extra moving though, and this motion is where these additional accelerations come into play.

Although playing catch while sky diving is beyond the scope of this thesis, modeling a force on a satellite most certainly is not. Knowing a satellite's acceleration is of utmost importance when modeling the satellites dynamics as discussed in Section 2.5. To calculate acceleration in an inertial frame simply find the force and divide by the mass using Equation 2.1. The inertial acceleration of an object could be found with information from a non-inertial reference frame by manipulating Equation 2.2 to yield Equation 2.3.

$$A_{obj} = \frac{F}{m} - (A_{rf} + 2\omega \times V_{rf} + \alpha \times R_{obj/rf} + \omega \times (\omega \times R_{obj/rf})) \tag{2.3}$$

Although the inertial acceleration could be calculated in a non-inertial reference frame it is considerably simpler to use an inertial reference frame for these calculations. This is typified when calculus is introduced into the equations as well. For example, determining an object's velocity with respect to time is attained by taking the derivative of $\frac{F}{m}$ with respect to time if the information is already in the inertial frame. But when considering a non-inertial frame of reference the time derivative of Equation 2.3 needs to be taken. This derivative would include the time derivative of each of those components. These additional calculations increase computing time and

are more difficult to accomplish when compared to working in an inertial reference frame. Thus, most individuals and programs execute these calculations in an inertial frame of reference.

2.2.1.1 Global Reference Frame. Now that the advantages of inertial reference frames have been highlighted, it is important to label an actual inertial reference frame to be used within this thesis. The trouble arises in that no known frame of reference is truly inertial because every known object in space has some acceleration. The earth rotates around the sun; the sun moves in the milky way galaxy; the galaxy is expanding in the universe. And while the center of the universe could be truly inertial, it is not particularly practical to measure everything from the center of the universe. Thus, one typically uses a frame of reference that is 'inertial enough' for their application. In regards to this thesis and to the SPHERES program at large, an 'inertial-enough' reference frame exists on-board the ISS and is referred to as the global reference frame. The global reference frame is considered 'inertial-enough' because while the ISS does move, the ISS does not move fast enough to generate accelerations with magnitudes large enough to produce noticeable errors within the six minute test periods of the SPHERES program. Thus, the terms global frame and inertial frame are considered to be the same for the purposes of this thesis. The origin of the global frame is in the center of the Unity module of the ISS [2]. The positive 'x' axis points to the front of Unity module; the positive 'y' axis points to the right. Lastly, the positive 'z' axis completes the right-hand coordinate frame by pointing down to the deck.

2.2.2 Body Frame. Although the inertial or global reference frame is used for complex calculations, a non-inertial reference frame is useful when working with a number of sensors and actuators specific to each spacecraft. A body reference frame unique for each satellite is useful when operating subsystems on the satellite because it allows users to manipulate these subsystems more intuitively. Thus, in regards to SPHERES, a body reference frame is used when working with the thrusters and the

controllers that direct how the thrusters fire within this project. The origin of the body frame of each satellite is located at the geometric center of each SPHERES. The positive 'x' axis points in the direction of the expansion port, while the positive 'z' axis points towards the pressure system regulator knob. Lastly, the positive 'y' axis completes the right-hand coordinate system [2]. Figure 2.1 illustrates how the body frame is aligned with the physical features of each satellite.

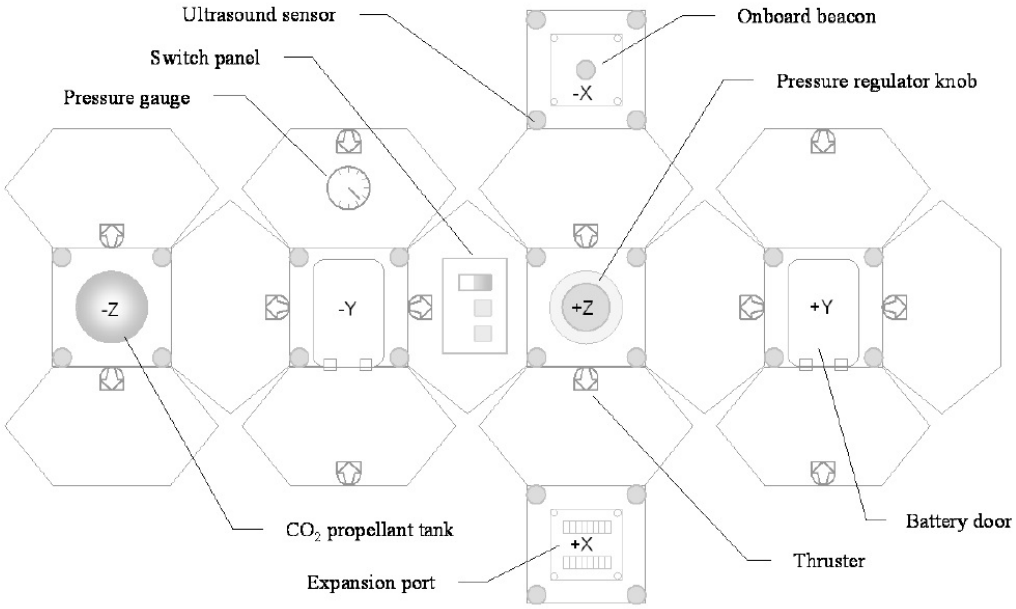

Figure 2.1: Unwrapped View of SPHERES Showing Body Frame Coordinate System & Physical Features [2]

2.3 Coordinate Rotations

The previous sections described the used of different coordinate frames and mentioned that each frame has specific advantages. This section discusses how to rotate vectors between coordinate frames, so that the most advantageous frame can be used. It is important to note that rotating a vector from one frame to another does not actually change the inertial vector, just the basis of a vector [30]. While the magnitude and true direction of a vector remain constant through the rotation, the direction of a vector appears to change as one observes the vector from a different

frame of reference. Generally speaking vectors in one frame can be rotated into another frame through the use of a direction cosine matrix, or 'rotation matrix'. Section 2.3.1 discusses what type of properties these rotation matrices have. A number of methods exist to calculate these rotation matrices, but three methods used to perform coordinate rotations are discussed: Euler angles, principle axis of rotation, and quaternions. Sections 2.3.2, 2.3.3, & 2.3.4 respectively discuss these different methods of rotations. The research with this thesis ultimately uses quaternions to handle coordinate rotations to prevent singularities. However, the other methods are used to transition user inputs into quaternions since the author does not expect most users to be able to intuitively insert desired quaternion vectors (see Section 3.3.1). Finally, Section 2.3.5 briefly discusses how to rotate different quaternions as needed to determine the orientation error in Section 3.1.3.

2.3.1 Direction Cosine Matrices. A rotation matrix is used in order to transform a vector from one frame to another by changing the basis of that vector. Consider the vector 'V' in Figure 2.2 as well as x-y-z and x'-y'-z' coordinate frames.

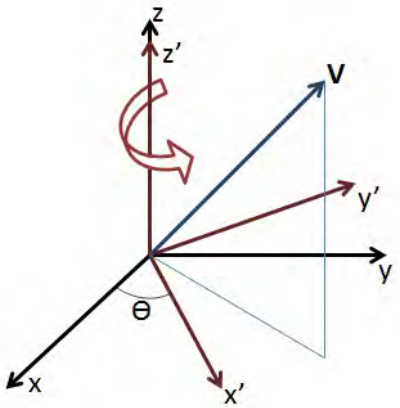

Figure 2.2: Illustration of a Rotation About 3^{rd} Axis

The two coordinate frames share the same third axis but the first axes and the second axes of the frames are separated by an angle θ. Next, assume the x-y-z frame to have three unit vectors 'a', 'b', & 'c' along each of the frames' axes, and imagine the x'-y'-z' frame has three unit vectors 'd', 'e', & 'f' along its axes as well. With this in mind the vector 'V' can be represented in the x-y-z frame through Equation 2.4, and can be related in the x'-y'-z' coordinate frame using Equation 2.5 [30].

$$V = V_a\hat{a} + V_b\hat{b} + V_c\hat{c} \tag{2.4}$$

$$V = V_d\hat{d} + V_e\hat{e} + V_f\hat{f} \tag{2.5}$$

While the magnitude of V remains the same regardless of which equation is first used, the components of each equation may be different because the vector is represented with a different basis associate with the two frames illustrated in Figure 2.2. In order to rotate from one to the other a relationship needs to be established between the two coordinate frames. With knowledge of how the coordinate frames are oriented with respect to each other the vector information in one frame can be converted to the other frame. Recalling that the coordinate frames of Figure 2.2 are separated by a third axis rotation of θ, the 'V' represented in the x-y-z frame can be written in the x'-y'-z' frame as shown in Equations 2.6-2.8.

$$V_d = cos(\theta) \cdot V_a + sin(\theta) \cdot V_b + 0 \cdot V_c \tag{2.6}$$

$$V_e = -sin(\theta) \cdot V_a + cos(\theta) \cdot V_b + 0 \cdot V_c \tag{2.7}$$

$$V_f = 0 \cdot V_a + 0 \cdot V_b + 1 \cdot V_c \tag{2.8}$$

Equations 2.6-2.8 can be written compactly in the form of a matrix as in Equation 2.9. This matrix is referred to as a direction cosine matrix or a rotation matrix.

$$V_{x'y'z'} = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} V_{xyz} \tag{2.9}$$

Equation 2.9 provides the rotation matrix to convert information out of the x-y-z frame and into the x'-y'-z' frame. To convert information the other direction the inverse of the rotation matrix is needed. Fortunately, one does not need to calculate this matrix by actually taking an inverse as this process tends to be computationally expensive. Recall that rotation matrices are based on the orientations between coordinate frames and that all the coordinate frames used within this thesis consist of a right-handed system of three orthogonal unit vectors. Therefore any rotation matrix for any pair of coordinate frames has orthogonal rows and columns that each represent unit vectors [31]. This means that any rotation matrix is orthonormal which is useful because the inverse of an orthonormal matrix is simply the transpose of that matrix [31].

The same process can be used to create rotation matrices about other axes as well. In particular, a rotation matrix can be created to correspond with each of the three orthogonal axes. A positive rotation about the first axis or 'x-axis' would result in the direction cosine matrix of Equation 2.10. Furthermore, a positive rotation about the 'y-axis', or 2-rotation would result in Equation 2.11, and a positive 3-rotation would result in Equation 2.12.

$$\mathbf{R}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix} \tag{2.10}$$

$$\mathbf{R}_3 = \begin{bmatrix} cos(\beta) & 0 & -sin(\beta) \\ 0 & 1 & 0 \\ sin(\beta) & 0 & cos(\beta) \end{bmatrix} \tag{2.11}$$

$$\mathbf{R}_3 = \begin{bmatrix} cos(\gamma) & sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

2.3.2 Euler Angles. Using Euler angles to assemble a series of rotation matrices allows for simplistic visualization of complex rotations. This is done by breaking up a rotation into a series of three simple rotations. The first rotation can be about any axis, while the second rotation is about either of the two axes yet to be used. Lastly, the third rotation is about either of the two axis not used for the previous rotation [32]. This is commonly referred to as a body-axis rotation since the angles build off of each other and stay with the rotating frame. An example Euler angle rotation sequence is the 3-2-1 sequence used by aircraft known as roll (ϕ),pitch (θ), and yaw (γ). Figure 2.3 describes how one frame is rotated using the roll pitch yaw sequence.

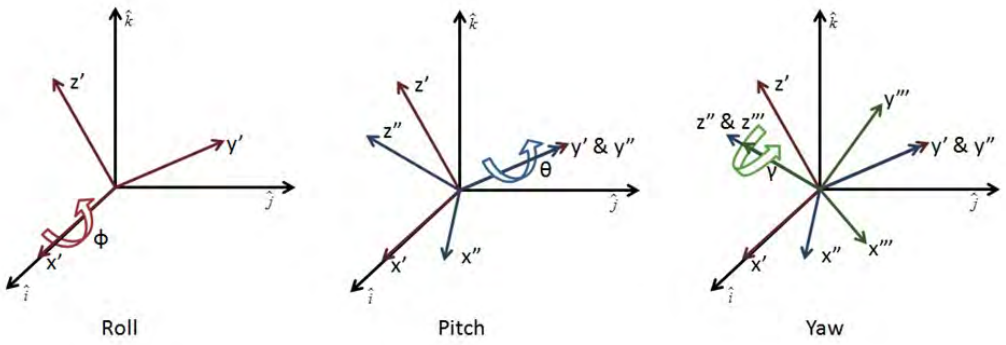

Figure 2.3: Illustration of Euler Angles Rotate Body Coordinate Frame

To understand Figure 2.3, let the $\hat{i}, \hat{j}, \hat{k}$ frame represent an inertial reference frame, and let the final orientation of the body frame be represented with the x"', y"', z"' frame. Both the body frame and the inertial frame start and the same spot

as shown on the left of Figure 2.3. To achieve a desired orientation the body frame rolls about its x-axis to reach x', y', & z'. The intermediate rotation is achieved by pitching about the second axis (pitch) of the x', y', z' frame to reach the x", y", z" frame. The body frame is then put into the desired orientation with the roll about the third axis (yaw) to reach the x"', y"', z"' frame. This is referred to as a 3-2-1 rotation because the body frame is first rotated about the first axis, then the second, and finally rotated about the third axis to reach the final orientation. The reason the numbers appear backwards has to do with matrix multiplication. Each consecutive rotation is pre-multiplied to the previous one as in Equation 2.14. Thus when read left to right the final rotation matrix consists of a 3, 2, and 1 rotation.

$$\mathbf{R}_{bi} = [\mathbf{R}_3(\gamma)][\mathbf{R}_2(\theta)][\mathbf{R}_1(\phi)] \tag{2.13}$$

$$\mathbf{R}_{bi} = \begin{bmatrix} cos(\gamma) & sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix} \tag{2.14}$$

Using the elementary direction cosine matrices these rotations can be used to generate the rotation matrix to convert vectors from the inertial frame to the body frame ($\mathbf{R}_{bi}$). The product of Equation 2.14 is displayed in Equation 2.16. The sine and cosine functions of Equation 2.16 are represented as s(x) and c(x) respectively.

$$EulerAngles = \begin{bmatrix} \phi \\ \theta \\ \gamma \end{bmatrix} \tag{2.15}$$

$$\mathbf{R}_{bi} = \begin{bmatrix} c(\theta)\cdot c(\gamma) & c(\phi)\cdot s(\gamma) + s(\phi)\cdot s(\theta)\cdot c(\gamma) & s(\phi)\cdot s(\gamma) - c(\phi)\cdot s(\theta)\cdot c(\gamma) \\ -c(\theta)\cdot s(\gamma) & c(\phi)\cdot c(\gamma) - s(\phi)\cdot s(\theta)\cdot s(\gamma) & s(\phi)\cdot c(\gamma) + c(\phi)\cdot s(\theta)\cdot s(\gamma) \\ s(\theta) & -s(\phi)\cdot c(\theta) & c(\phi)\cdot c(\theta) \end{bmatrix} \tag{2.16}$$

In addition to creating a rotation matrix from a series of Euler angles, Equation 2.16 can also be used to find a set of Euler angles when given a rotation matrix. $\mathbf{R}_{bi}$ can be used to back out the Euler angles as is done in Equations 2.17- 2.19.

$$\theta = sin^{-1}(R_{3,1}) \tag{2.17}$$

$$\phi = sin^{-1}(\frac{-R_{3,2}}{cos(\theta)}) \tag{2.18}$$

$$\gamma = sin^{-1}(\frac{-R_{2,1}}{cos(\theta)}) \tag{2.19}$$

A problem arises with this method when the frame pitches $\pm 90^{\circ}$ because the roll and pitch angles can not be identified. This type of singularity is not unique to the 3-2-1 rotation. In fact, all Euler angle rotations encounter a singularity when the second rotation causes the first and third rotations to become mathematically indistinguishable. This is because an infinite amount of Euler angles could be generated from the rotation matrix when a singularity exists. Therefore, controllers are unable to reliably produce accurate Euler angles when these singularities are present. Two options exist to work around this. Either the user could exercise caution to ensure the controller is never faced with a singularity by constraining the inputs and limiting the frame orientation, another approach could be used to determine rotation matrices. The latter option is chosen for the SPHERES application since it is impractical to limit satellite orientations.

2.3.3 Eigenaxis of Rotation & Principal Euler Angle. The latter option leads to the use quaternions as discussed in Section 2.3.4. Although the SPHERES program uses quaternions to perform coordinate rotations, quaternions in and of themselves are difficult to understand without knowledge of another rotation method. Thus, the discussion of an eigenaxis of rotation and it associated principal Euler angle is included to provide a link between rotating with Euler angles and rotating with quaternions.

The eigenaxis approach uses one rotation about a single axis with one angle instead of three axes with three angles. This method is derived from Euler's theorem that the displacement of a rigid body with one point fixed is a rotation about some axis [33]. Instead of combining three simple rotations to describe a complex rotation as is done with the method of Euler angles, this approach describes an arbitrary rotation by rotating the coordinate frame about one stationary axis [33]. Although only one axis and one angle are needed to perform this rotation, the axis of rotation may not line up with one of the principal axes of the coordinate frame. Thus, this rotation vector, or eigenaxis of rotation needs to be calculated along with the principal Euler angle. Wie provides the derivation for this process [32]. For the purposes of SPHERES and this thesis, the eigenaxis (ε) is found through the cross product of two vectors as shown in Figure 2.4.

The first vector (red) is where the satellite is supposed to point, and the second vector (blue) dictates where the satellite is currently pointing. The cross product produces the vector that the satellite needs to rotate about to get to the desired orientation. Furthermore, the principal Euler angle (Φ) is found using the dot product of those two vectors as shown in Equation 2.20.

$$\Phi = acos(\frac{\bar{a} \cdot \bar{b}}{\mid \bar{a} \mid\mid \bar{b} \mid}) \tag{2.20}$$

With an understanding of how this method is derived, it is useful to know how to create these values when given a rotation matrix. Equation 2.21 shows how to derive the principal Euler angle (Φ) from a given rotation matrix, $\mathbf{R}$. '$trace(\mathbf{R})$' is

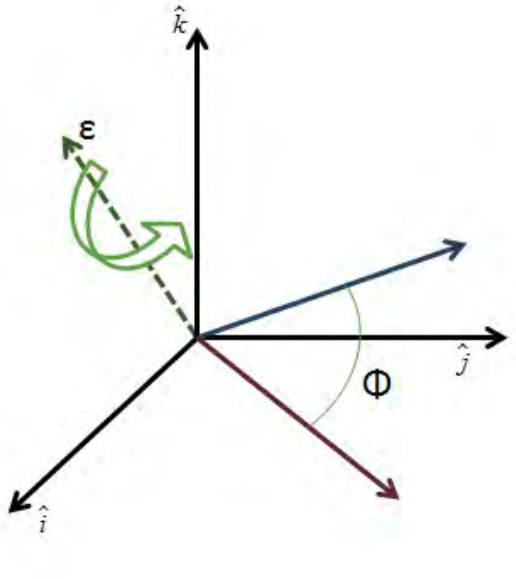

Figure 2.4: Illustration of Eigenaxis of Rotation

the trace operation applied to matrix **R**, which is the sum of the diagonal elements of **R**. In addition, Equation 2.22 demonstrates how to determine the skew-symmetric representation of the eigenaxis of rotation (ε) using rotation matrix **R** and angle Φ. The true representation of ε is taken from its skew matrix using Equation 2.23.

$$\Phi = cos^{-1}[\frac{1}{2}(trace(\mathbf{R}) - 1)] \tag{2.21}$$

$$\varepsilon^x = \frac{1}{2sin(\Phi)}(\mathbf{R}^T - \mathbf{R}) \tag{2.22}$$

$$\varepsilon^x = \begin{bmatrix} 0 & -\varepsilon_3 & \varepsilon_2 \\ \varepsilon_3 & 0 & -\varepsilon_1 \\ -\varepsilon_2 & \varepsilon_1 & 0 \end{bmatrix} \tag{2.23}$$

Unfortunately, eigenaxis rotations also encounter a singularity. This occurs when the rotation matrix is an identity matrix which means no rotation is necessary. When this happens, $\Phi = 0$ which results in ε becoming undefined as Equation 2.22 can

not be evaluated [33]. One mechanism by which this singularity may be avoided is the inclusion of a cleverly chosen fourth parameter which leads into the implementation of quaternions.

2.3.4 Quaternions. In order to perform multiple coordinate rotations without risk of singularity Euler parameters, otherwise known as quaternions, are used. The quaternion vector, $\bar{q}$, uses four components to represent orientations in three-dimensions.

$$\bar{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{2.24}$$

Similar to the eigenaxis method, quaternions also determine coordinate orientations through a single axis of rotation [32]. With this in mind, the first three quaternions ($\tilde{q}$) are related to the eigenaxis. The fourth quaternion (q_4) is included to prevent a singularity when no rotation occurs. This naturally leads one to represent quaternions with respect to the eigenaxis and principal Euler angle as in Equations 2.26 and 2.27.

$$\tilde{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{2.25}$$

$$\tilde{q} = \bar{\varepsilon} sin(\frac{\Phi}{2}) \tag{2.26}$$

$$q_4 = cos(\frac{\Phi}{2}) \tag{2.27}$$

In addition to these equations, the four quaternions values are not independent of one another. Instead, they are constrained by the relationship presented in Equation 2.28 [32]. The reason for this constraint derives from the fact that quaternions are based on the eigenaxis (ε), and the root sum square of the eigenaxis is also equal to one.

$$\tilde{q}^T \tilde{q} + {q_4}^2 = {q_1}^2 + {q_2}^2 + {q_3}^2 + {q_4}^2 = 1 \tag{2.28}$$

Thus, the four values that make up a quaternion can be thought of as a four dimensional array that represents the three dimensional eigenaxis. The inclusion of the fourth term prevents any singularities from occurring. Recall, that the eigenaxis method for rotations encounters a singularity when no rotation is needed between reference frames ($\Phi = 0$). In terms of the quaternions, when $\Phi = 0$, $q_4 = 1$ and $\tilde{q} = \bar{0}$. This avoids the singularity because a unique quaternion vector exists for each orientation. Therefore, control algorithms can use quaternion vectors solely without losing knowledge of the satellite's orientation. A rotation matrix is determined with quaternions using Equation 2.29. The subscript 'bi' denotes the rotation matrix $\mathbf{R}$ is used to convert information from the inertial frame to the body frame. In addition, $\mathbf{I}$ is an identity matrix of rank three.

$$\mathbf{R}_{bi} = ({q_4}^2 - \tilde{q^T}\tilde{q})\mathbf{I} + 2\tilde{q}\tilde{q}^T - 2q_4\tilde{q}^x \tag{2.29}$$

2.3.5 Rotating Quaternions. Quaternions have been shown to successfully handle coordinate rotations between two reference frames, and these reference frames allow for 3x1 column vectors of information to be converted from one frame into another. But the quaternion vector consists of a four terms and not three. Thus, a specific method needs to be included to describe how to rotate quaternions and retain information within the vector. This is particular necessary to determine the quaternion error when the desired quaternions are present.

To illustrate how to handle successive rotations with quaternions consider three coordinate frames: the 'x' frame, the 'y' frame, and the 'z' frame. Let $\bar{q}'$ be associated with the rotation matrix from the 'x' to the 'y' frame, or $\mathbf{R}^{yx}$, $\bar{q}''$ is describes how to rotate from the 'y' to the 'z' frame ($\mathbf{R}^{zy}$), and $\bar{q}$ correspond to the rotation matrix from the 'x' to the 'z' frame ($\mathbf{R}^{zx}$). In regards to rotations matrices, $\mathbf{R}^{yx}$ and $\mathbf{R}^{zy}$, could be used to find $\mathbf{R}^{zx}$ by taking the product of the first two direction cosine matrices. The difficulty with that method arises when one attempts to keep track of each of the variables for each of those direction cosine matrices. Each rotation matrix consists of nine values, six of which are independent. Quaternions only require knowledge of four values, and only three are independent. The equations to describe the relationship between the quaternions for the different reference frames are shown in Equations: 2.30 and 2.31 [34].

$$\tilde{q} = q_4''\tilde{q}' + q_4'\tilde{q}'' + \tilde{q}' \times \tilde{q}'' \tag{2.30}$$

$$q_4 = q_4'q_4'' - (\tilde{q}')^T(\tilde{q}'') \tag{2.31}$$

To simplify these equations one can redefine these relationships into matrix form as in Equations: 2.32 and 2.33.

$$\bar{q} = \begin{bmatrix} q_4'' & q_3'' & -q_2'' & q_1'' \\ -q_3'' & q_4'' & q_1'' & q_2'' \\ q_2'' & -q_1'' & q_4'' & q_3'' \\ -q_1'' & -q_2'' & -q_3'' & q_4'' \end{bmatrix} \bar{q}' = \mathbf{M}(q'')\bar{q}' \tag{2.32}$$

$$\bar{q} = \begin{bmatrix} q_4' & -q_3' & q_2' & q_1' \\ q_3' & q_4' & -q_1' & q_2' \\ -q_2' & q_1' & q_4' & q_3' \\ -q_1' & -q_2' & -q_3' & q_4' \end{bmatrix} \bar{q}'' = \tilde{\mathbf{M}}(q')\bar{q}'' \tag{2.33}$$

Both $\mathbf{M}$ and $\tilde{\mathbf{M}}$ are orthonormal which means that the inverse of these matrices is simply its transpose [31]. Thus, in addition to simplifying the equations into matrices, the matrix form of the equation is computationally more efficient when inverses are required for the transmuted quaternion matrix, or $\tilde{\mathbf{M}}$. This is beneficial as $\tilde{\mathbf{M}}^{-1}$ is typically used to determine the quaternion error when the desired quaternions are present.

2.4 Control Techniques

Coordinate rotations and quaternions play a large role in how to understand and operate a satellite. Yet the core issue is to ensure the satellite operates as instructed. That is to mean the satellite must meet required pointing accuracies, be at the right place at the right time, and performing the way the satellite was designed to. This is accomplished through the application of controls. Although control techniques date back to the mid 1800s, the study of control theory did not gain momentum until the early and mid 1900's with the study of flight and fire-control systems [35]. Control techniques are split up into two broad sections. Control theory exists for both linear and non-linear systems. Linear systems must satisfy two properties, namely additivity and homogeneity. For example, the function f is said to have the property of additivity if $f(x) + f(y) = f(x + y)$. Furthermore, homogeneity means the the function is closed under scaler multiplication. Thus if c is a scaler the function f would be closed under scalar multiplication if $f(cx) = c \cdot f(x)$ [35]. In addition to linear and non-linear systems, control techniques can be applied to optimize a cost functional or take advantage of a particular aspect of a system. This section discusses various control techniques that are used throughout the design of the speed and path control algorithm.

2.4.1 Linear Stability. The primary function of a control system is to ensure system stability. In terms of linear systems, three broad terms exist to classify the stability of a linear system. A linear system can either be unstable, marginally stable,

or asymptotically stable [36]. A marginally stable system is usually undesirable because while the system response remains in the vicinity of the commanded response, a marginally stable system is unable match the commanded response like an asymptotically stable system can. Stable linear systems are also classified by the order of the response. A first-order system only has one pole while a second-order system has two poles. Higher order systems can typically be represented as second-order systems [37]. Most control applications design systems to be second-order systems because these types of systems can be 'tuned' to meet a wide range of different desires.

The generic format for a second-order transfer function is shown in Equation 2.34. In this equation, ζ is the damping coefficient and ω_n is the natural frequency of the system.

$$TF = \frac{1}{s^2 + 2\zeta\omega_n s + {\omega_n}^2} \tag{2.34}$$

Changing ζ reflects how much damping is present in the system. This in turn, can drastically change the system performance. When $\zeta = 0$ the system is marginally stable because the signal never damps out. If the $\zeta > 1$ the system is said to be over-damped because the system has so much damping that the system takes a significant amount of time to reach the commanded position. Both un-damped and over-damped systems are undesirable for most engineering purposes. Most systems are designed to be under-damped and have the property $0 < \zeta < 1$. This is because the transient response of under-damped systems typically contain the fastest characteristics. Lastly, when $\zeta = 1$ the linear system is considered to be critically-damped. This occurs as the second order poles transition from an under-damped system to an over-damped system. When the two poles are under-damped they exist on the left hand side of the complex plane and have symmetry about the negative real axis. Over-damped poles lie at different points on the negative real axis. Critically-damped poles exist on the same spot of the negative real axis of the complex plane. Although knowledge of the pole locations for a system is important, this knowledge is a little abstract.

Another way to observe how damping affects a second-order systems is by observing the transient response of the system. Figure 2.5 illustrates how the transient response of second-order systems are affected by ζ.

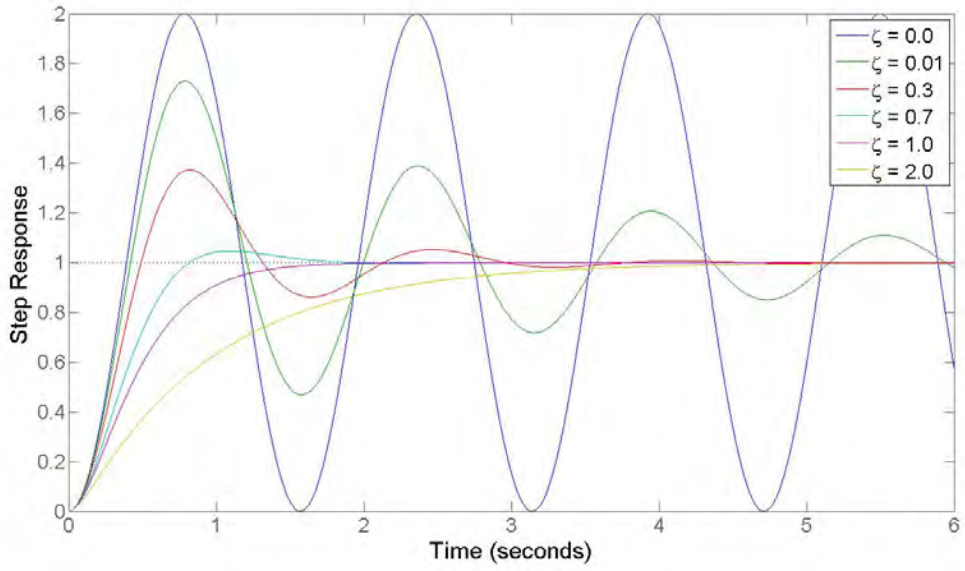

Figure 2.5: Example of Second-order Responses

When considering the systems transient response, a number of methods are used to objectively compare system performance. Three in particular are used within this thesis: rise time, settling time, and percent overshoot. The rise time of a system is the time needed for the signal to transition from the initial displacement to the final value. Two methods are typically used to determine this. One can either record the time from the waveform's initial displacement to the waveform's final value, or one can measure the time required for the signal to transition from 10% of the final value to 90% of the final value [37]. Settling time is used to measure how long it takes the signal's damped oscillations to reach and stay within $\pm$ 2% of the final value. Lastly, the percent overshoot measures how far the signal overshoots the final, or steady-state, value at the time when the signal is at the highest peak [36]. These parameters are useful metrics for comparing systems to each other and determining which response is the best. This is used to determine optimal gains in Chapter III.

Nyquist plots are also used to determine the stability of closed-loop systems. Nyquist plots use the system's open-loop frequency response and open-loop poles to provide the phase and gain margins of the system [36]. Stability is determined through the use of Equation 2.35 where 'Z' is the number of closed-loop zeros in the right-half plane, 'N' is the number of clockwise encirclements of the point (-1+j0), and 'P' is the number of unstable open-loop poles.

$$Z = N + P \tag{2.35}$$

In addition to determining system stability, Nyquist plots are also used to evaluate the system stability margins as well. Gain margins indicate how much open-loop gain can be applied to or taken out of a system before the system goes unstable. Phase margins indicate how much delay, or phase, can be applied to the open-loop system to make the closed-loop system unstable [37]. The phase margin of an arbitrary system is shown in Figure 2.6.

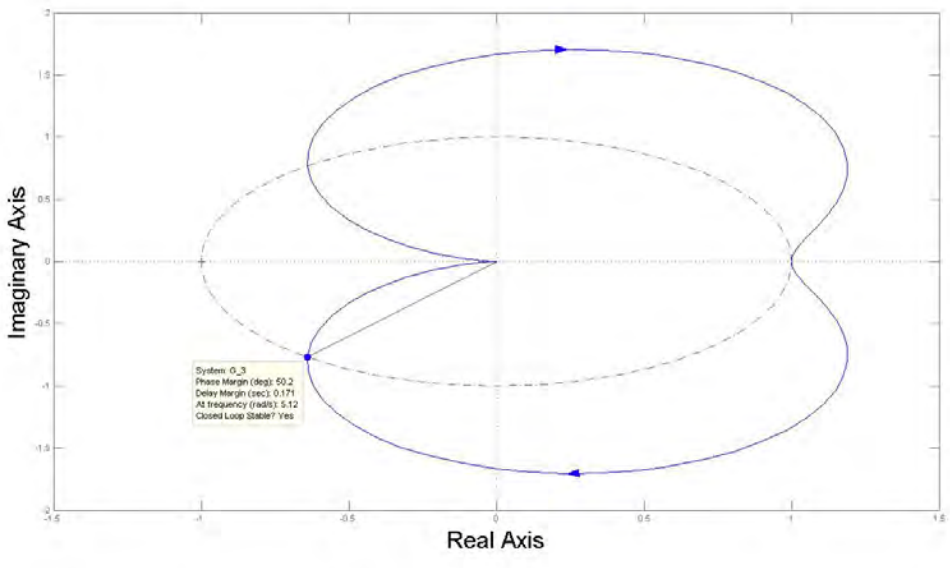

Figure 2.6: Example of Nyquist Plot

On the Nyquist plot the phase margin is determined by finding the angle between the negative real axis and the point of the open-loop Nyquist response that is on the unit circle (represented by the dotted line). The gain margin is calculated as the difference between the point (-1+j0) and point of the open-loop response that intersects the negative real axis. In addition, gain margins are typically represented in decibels so this difference would need to be converted as well. The example system in Figure 2.6 has a 50° phase margin and an infinite gain margin.

2.4.2 Lyapunov Stability. As Section 2.5.2 describes, SPHERES' dynamics associated with the satellite's orientation is nonlinear. Thus, before a controller is designed, it is worth pausing to consider how to stabilize nonlinear systems. Additionally, since nonlinear systems can contain more complex and exotic behavior than their linear counterparts, it is worth considering what type of stability is desirable. Due to the complexity of nonlinear systems, stability is often determined using linear concepts about equilibrium states found within the system. An equilibrium state is any state within the nonlinear system that remains stationary for all time if the system starts at that equilibrium state. To define stability about an equilibrium point two regions should be mentioned. Let S_R consist of a region around the equilibrium state that is greater than zero. In addition, S_r is a subset of this region. Figure 2.7 illustrates how these regions are used to determine stability around a particular equilibrium state.

Stable systems are said to remain with region S_R so long as the trajectory begins within S_r. In essence stability (often referred to as Lyapunov stability) suggests that a state trajectory ($x(t)$) will remain in the vicinity of the equilibrium state if the trajectory begins sufficiently close to it. A system is said to be asymptotically stable if all state trajectories that begin within S_r return to the equilibrium state. In addition, a system is said to be globally asymptotically stable if both S_R and S_r contain the entire subspace [4].

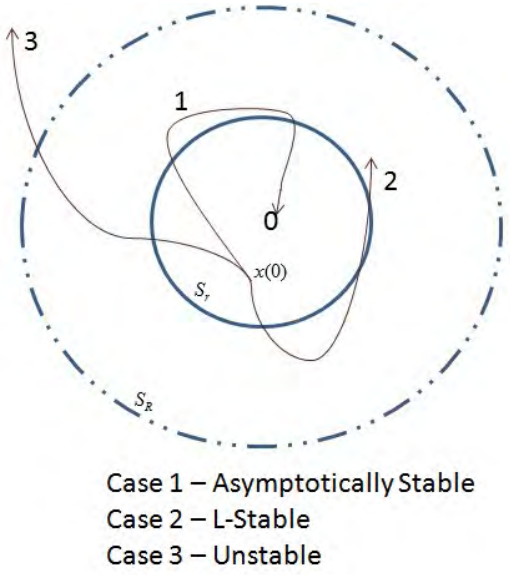

Figure 2.7: Concepts of Stability

For precise control of satellite orientation, the satellite dynamics are required to be asymptotically stable. In order to make a system asymptotically stable, an appropriate Lyapunov function needs to be derived to make the system perform as desired.

2.4.3 Lyapunov Functions. Lyapunov functions are used to determine the stability of a nonlinear system. By implementing the control law into the Lyapunov function as done in Section 3.2.4, Lyapunov equations can be used to stabilize the system. Lyapunov functions are defined by the properties carried within the function. If a function meets these properties it is considered to be a Lyapunov function. A Lyapunov function, V, is a scalar function with continuous partial derivative such that the function is positive definite and the rate of the function, $\dot{V}$, is negative semi-definite. If the system can be represented as a Lyapunov function then the system is at least Lyapunov stable. Furthermore, a system is found to be globally asymptotically stable if $\dot{V}$ is negative definite [4]. Thus, if a positive definite Lyapunov function with a negative definite rate can be applied to a particular nonlinear system, that system is globally asymptotically stable. Lyapunov functions can be used in this manner

to determine an appropriate control law to make a nonlinear system asymptotically stable. Yet before this can be done one should understand how to classify matrices as positive or negative definite.

2.4.3.1 Definite Matrices. The definiteness of a matrix allows one to understand what values that matrix will generate in quadratic form. An example of a quadratic function is shown in Equation 2.36.

$$F(\bar{x}) = \frac{1}{2}\bar{x}^T \mathbf{A} \bar{x} \tag{2.36}$$

$F(\bar{x})$ can either be positive, negative, or zero. Knowledge of the definiteness of a matrix allows one to know what values $F(\bar{x})$ can take on for any $\bar{x}$. A positive definite matrix indicates $F(\bar{x})$ will be a positive for all values of $\bar{x}$ except $\bar{x} = \bar{0}$ [38]. In addition, a negative definite matrix means $F(\bar{x})$ will be negative whenever $\bar{x} \neq \bar{0}$[2]. To determine whether a matrix is positive or negative definite one must calculate the eigenvalues of the matrix.

A positive definite matrix consists of only positive eigenvalues, and a negative definite matrix contains only negative eigenvalues [4]. Although other types of definite matrices exist, these two types are particularly useful for Lyapunov functions and for the design of the quaternion controller in Section 3.2.4. In addition to understanding general stability criterion for both linear and nonlinear systems, two specific types of control techniques are of particular use for commanding SPHERES: bang-bang control and the linear quadratic regulator.

2.4.4 Bang-Bang Control. SPHERES achieves any rotation and translation by firing combinations of its twelve cold gas thrusters [2]. These thrusters either fire at a specific value or they do not fire at all. This on-off discontinuity is often dealt with bang-bang controllers. Bang-bang controllers are a feedback controller that sharply

[2]When $\bar{x} = \bar{0}$ the matrix $\mathbf{A}$ has no affect of $F(\bar{x})$.

switches between two states, such as an on and off command. An example bang-bang controller is found within many household thermostats. If the temperature is below a specific value the heater is commanded to operate and raise the temperature. Likewise, if the temperature of the house is above a set temperature, the thermostat commands the air conditioning unit to cool the house. If the temperature value used to trigger the heater was the same value as the one used to trigger the cooling unit, either of the two units would be running at any given time. This is because anytime the temperature was not perfectly maintained the controller would command the appropriate unit to adjust the temperature. In order to prevent this a dead-zone is implemented to provide a gap in which neither unit is commanded to fix the temperature. This is discussed further in Section 3.2.6.1. Bang-bang control also uses previous knowledge of the states to operate. One way to do this is to consider the rate of change for the state as the actual state. In this fashion, the control designer has the ability to 'look-ahead' and predict how the states are going to change an adjust accordingly. This is analogous to approaching a stop sign when driving a car. When approaching a stop sign the driver considers how fast they are approaching the sign and applies the brakes as necessary. The same concept applies to bang-bang control. This concept is depicted by observing the relationship between the state and the rate of change of a state in the phase plane. Figure 2.8 illustrates how a system with bang-bang control would appear on a phase plane.

Figure 2.8 shows a one dimensional case for the relationship between the position and velocity error of a sample spacecraft. In this sample the spacecraft begins with a positive position and velocity. As time progresses the velocity error improves for a time and then becomes more negative to correct the position error. As this relationship passes the switching line the priority shifts to correct the position error. The controller switches between which error has correction priority based on where the relationship of position and velocity error is with respect to the switching line. Eventually, due to the nature of the bang-bang controller, the controller gets to a point where further improvements on the state errors result from switching correc-

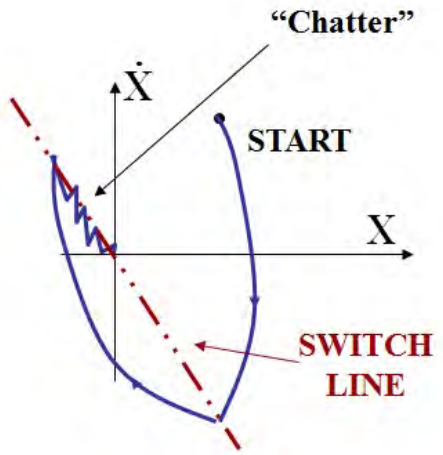

Figure 2.8: Example Phase Plane of Look Ahead Controller [3]

tion priority rather quickly. This results in an undesirable amount of fuel use and is known as chatter. A dead-zone is implemented to minimize the effects of chatter and is described in Section 3.2.6.1. Further use of bang-bang control is mentioned in Section 3.2.1 when the translational controller is developed. In addition to bang-bang control, the linear quadratic regulator is also considered to provide optimal control to SPHERES.

2.4.5 Linear Quadratic Regulators. One method for developing optimal control is the use of the Linear Quadratic Regulator (LQR). This form of control optimizes controller gains to minimize a quadratic cost function. So long as the quadratic cost function accurately reflects the designer's concept of 'good' performance, the LQR will provide the optimal gain to minimize the cost function and best reflect the user's desires [39]. LQR controllers perform well in conjunction with a Linear Quadratic Estimator (LQE), or Kalman filter, because the filter is able to compensate for random initial conditions and inputs (disturbances and measurements) corrupted with white noise. The union of the LQR and the LQE form the Linear Quadratic Gaussian controller which is the optimal control solution when noise is present throughout the system [39]. Since the SPHERES estimator handles the random disturbances,

LQR design is particularly useful in the application of controlling the linear portion of SPHERES.

In regards to SPHERES, the LQR minimizes the cost functional, J, shown in Equation 2.37. The cost function is split into two sections. Given an initial state error, the first part is used to evaluate the cost induced from the state error during the controller's operation. The second term in the integrand is used to penalize how much control is required to return the system to the nominal state.

$$J(x(t), u(t)) = \frac{1}{2}\int_0^\infty \{x^T(t)\mathbf{Q}x(t) + u^T(t)\mathbf{R}u(t)\}dt \tag{2.37}$$

The weighting matrix $\mathbf{Q}$ is applied to minimize the state error while the matrix $\mathbf{R}$ is selected to minimize control usage. Both matrices are picked by the designer and are typically diagonal matrices. Higher values of each mean a higher cost is assigned to that portion of the solution. The $\mathbf{Q}$ and $\mathbf{R}$ matrices are used to balance perfect performance against system efficiency. If a designer desired to have minimal error on the states they would use a $\mathbf{Q}$ that was relatively larger than $\mathbf{R}$. This would force the LQR to sacrifice control usage to meet the demand of small error. Likewise, if if there exists a desire to conserve control usage to save fuel usage or extend the lifetime of a system, larger emphasis needs to be placed on minimizing control usage by assigning a larger $\mathbf{R}$. Regardless of the designer requirements, the importance of $\mathbf{Q}$ and $\mathbf{R}$ is not in the actual magnitudes within matrices. Instead the designer should be concerned with the ratio between $\mathbf{Q}$ and $\mathbf{R}$. This is because the value of J is arbitrary and only useful to compare solutions with the same $\mathbf{Q}$ and $\mathbf{R}$ matrices. Certainly, if large $\mathbf{Q}$ and $\mathbf{R}$ weights were used the cost function would be high. But J would be high for any of the solutions with the same $\mathbf{Q}$ and $\mathbf{R}$. Because J is linear in $\mathbf{Q}$ and $\mathbf{R}$, the optimal solution to the cost function with the large $\mathbf{Q}$ and $\mathbf{R}$ would be the same solution as the optimal one for a different $\mathbf{Q}$ and $\mathbf{R}$ so long as the ratio between each of the $\mathbf{Q}$ and $\mathbf{R}$ matrices was preserved [39].

Not only does the LQR minimize the cost functional of Equation 2.37, but the LQR also produces the optimal steady-state gain matrix, $\mathbf{K}$. This is achieved through the use of Equation 2.38.

$$\mathbf{K} = \mathbf{R}^{-1}\mathbf{B_u}^T\mathbf{P} \tag{2.38}$$

The optimal steady-state gain matrix is generated with the LQR control weight, the control input matrix and $\mathbf{P}$. The matrix $\mathbf{P}$ is the solution to the algebraic Riccati equation shown in Equation 2.39. The algebraic Riccati equation is found by setting the Riccati differential equation equal to zero [39]. Once $\mathbf{P}$ is determined, the optimal feedback gain is calculated via Equation 2.38.

$$\mathbf{PA} + \mathbf{A}^T\mathbf{P} - \mathbf{PB_u}\mathbf{R}^{-1}\mathbf{B_u}^T\mathbf{P} + \mathbf{Q} = 0 \tag{2.39}$$

Another advantage to the LQR is that this controller is particularly robust. The linear quadratic regulator is guaranteed to have a upside gain margin of ∞ and a downside gain margin $\leq \frac{1}{2} = -6dB$. In addition, the LQR ensures the system has a phase margin $\geq 60^\circ$ [35]. The stability margins from the LQR can also be used to provide a sort of best case for nonlinear systems as mentioned in Section 3.2.1.

The control techniques discussed throughout this section are used in the design of the speed and path control algorithm. Before designing the controller the plant must be fully characterized as described next.

2.5 Modeling SPHERES Plant

While the majority of this thesis is dedicated to describe the development of the speed and path control algorithm, the SPHERES plant is also constructed for simulation in this thesis. The description of the plant is developed based on information supplied. The plant is a crucial piece of the simulation, but the design of the plant is not the focus of this research and is therefore included in this background chapter.

Nonetheless, the author believes the control algorithm can best be understood when the plant that the algorithm is supposed to control is fully appreciated. Figure 2.9 illustrates how the SPHERES plant is divided for the purposes of this discussion as well as within simulation.

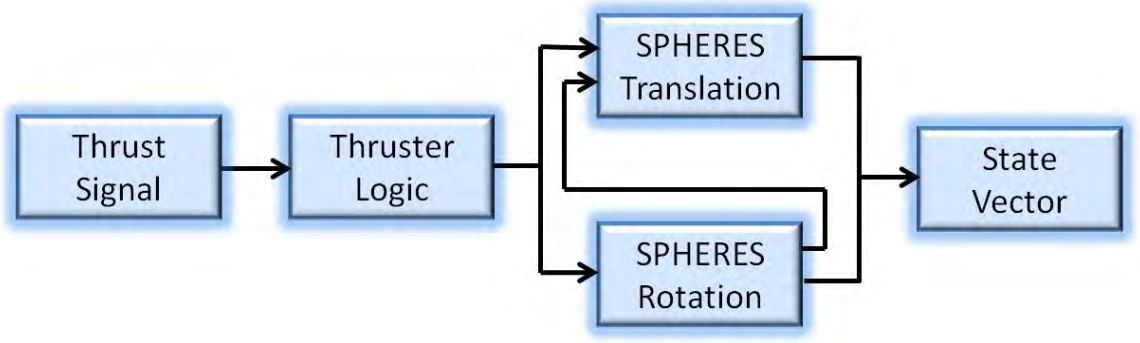

Figure 2.9: Simple Diagram of Plant

Due to the thruster spatial arrangement (as seen in Table 2.1), the system dynamics determining how each SPHERES translates and rotates are mostly decoupled. This means that the SPHERES plant can be divided into two components. One relates to the translational components of SPHERES, and the other relates to the rotational components of SPHERES. Section 2.5.1 describes how the rates of the SPHERES position and velocity are modeled. Section 2.5.2 illustrates how to determine the quaternions and the angular rates to understand the satellite orientation. Lastly, Section 2.5.3 specifies how the thrusters are interpreted into corresponding torques and forces applied to the satellite.

Additionally, it should be noted that the only relevant external force applied to the satellite is the force generated from the thrusters. Other forces typically known to affect relative satellite motion result from J_2, atmospheric drag, and the approximation of the orbital motion described by Kepler shown through the Clohessy-Wiltshire equations. These effects are ignored for a number of reasons. Air drag does affect SPHERES operating inside the ISS however, for this application due to the relatively low speeds, drag is significantly smaller than the force generated by the thrusters [2]. Thus, it is ignored for simplicity within this simulation since any drag in real tests

can be thought of as a disturbance which can easily be overcome with the thrusters[3]. Furthermore, the longest run time of any SPHERES test inside the ISS has been six minutes, but the tests typically run two to four minutes in length. In this time period, orbital motion described by the Clohessy-Wiltshire equations cause a formation of two satellites to transition approximately five centimeters. This results in a $0.2\frac{mm}{s}$ change in velocity. Since the SPHERES thrusters produce a change in velocity approximately fifty times greater, the orbital motion approximated by the Clohessy-Wiltshire equations have been ignored. The precession of SPHERES' orbital plane due to the oblateness of the Earth has also been neglected in the plant model. This can be ignored since all of the coordinate frames are, for all intents and purposes, equally affected by J_2 for the duration of SPHERES tests.

2.5.1 Position & Velocity Model. The translation component of SPHERES consists of a position and a velocity vector. Since Newton's Second Law is the fundamental equation governing how the satellite translates, the position and velocity states are modified in the global reference frame. Equation 2.40 identifies how the position and velocity vectors are identified in the state vector of the SPHERES' plant.

$$\bar{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} X & Position \\ Y & Position \\ Z & Position \\ X & Velocity \\ Y & Velocity \\ Z & Velocity \end{bmatrix}_{Global\ Frame} \tag{2.40}$$

Newton's Second Law (Equation 2.1) is linear in the global frame the translational component of the state vector, $\bar{X}$, is a linear system as well. Thus, the rate of the state vector ($\dot{\bar{X}}$) is represented in state-space form in Equation 2.41.

[3]Ignoring air drag does mean the simulation does not exactly match reality. Thus, future research could be done to make this simulation a better model of reality by including air drag effects.

$$\dot{\bar{X}} = \mathbf{A}\bar{X} + \mathbf{B}\bar{u} \tag{2.41}$$

$$\bar{Y} = \mathbf{C}\bar{X} + \mathbf{D}\bar{u} \tag{2.42}$$

The **A** matrix displays how the states affect one another and the **B** matrix identifies how the control affects the system. As previously stated, this thesis does not consider the estimator and assumes the controller has access to full-state feedback, and that the control has no direct affect on the measurement $\bar{Y}$. Therefore the output $\bar{Y}$ is $\bar{X}$. In order to determine the values in matrices **A** and **B**, all the forces need to identified on SPHERES, which in this case are the forces generated from the thrusters. This results in the following state-space matrices for Equation 2.41:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.43}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1/m & 0 & 0 \\ 0 & 1/m & 0 \\ 0 & 0 & 1/m \end{bmatrix} \tag{2.44}$$

The rate at which the position changes is directly a result of what the satellite's velocity is. In addition, the rate of change of the velocity is only a function of the force generate by the thrusters and the mass of the satellite. Recall that the state-space

matrices update states in the global frame though. As a result, the forces generated from the thrusters must be rotated from the body frame to the global frame. This is accomplished using the quaternions to generate a rotation matrix via Equation 2.29. Since the position and velocity of SPHERES is used to determine where one satellite is relative to the other, the term translational states is used to quickly refer to the first six states of the SPHERES state vector which is the position and the velocity of the SPHERES satellite.

2.5.2 Model for Quaternions & Angular Rates. The rotational component of the plant model consist of the quaternions and angular rates ($\bar{\omega}$). Although there are four quaternions, only three angular rates exist. These values correspond to how fast the body frame axes are rotating. These values make up the last seven of the thirteen values in the state vector shown in Equation 2.45, and are referred to as the rotational states.

$$\bar{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_10 \\ x_11 \\ x_12 \\ x_13 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ \dot{X} \\ \dot{Y} \\ \dot{Z} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ \omega_X \\ \omega_X \\ \omega_X \end{bmatrix}_{Global\ Frame} \tag{2.45}$$

The rate of change for the quaternions are split into two equations. The first three quaternions ($\tilde{q}$) are governed by Equation 2.46 while the fourth quaternion is regulated by Equation 2.47 [32].

$$\dot{\tilde{q}} = \frac{1}{2}(q_4)\bar{\omega} - \omega^x \tilde{q} \tag{2.46}$$

$$\dot{q}_4 = -\frac{1}{2}\bar{\omega}^T \tilde{q} \tag{2.47}$$

The term ω^x is the skew symmetric representation of the angular rates. Equations 2.46 and 2.47 show that the quaternions depend on that angular rates of the body frame axes. The equation for the angular rates are derived from the applied external moments generated from the thrusters. The external moments, $\bar{M}$, are equivalent to the change in angular momentum, $\dot{\bar{H}}$ as shown in Equation 2.48 where the term **MOI** represents the satellite's mass moment of inertia.

$$\bar{M} = \dot{\bar{H}} = \mathbf{MOI}\dot{\bar{\omega}} + \omega^x \mathbf{MOI}\bar{\omega} \tag{2.48}$$

Furthermore, for the application to SPHERES, the only external moments applied on the satellite come from the thrusters. This results in Equation 2.49 where the term $\bar{u}$ is a 3x1 vector containing the applied torques derived from the thrust profile.

$$\bar{M} = \bar{u} \tag{2.49}$$

Combining Equation 2.48 and Equation 2.49 results in the rate of $\bar{\omega}$, as shown in Equation 2.50.

$$\dot{\bar{\omega}} = \mathbf{MOI}^{-1}(\bar{u} - \omega^x \mathbf{MOI}\bar{\omega}) \tag{2.50}$$

Thus, with Equations 2.41, 2.46, 2.47 and 2.50, the entire state vector can be determined for any point in time assuming one has knowledge of the current state vector, the physical properties of the satellite, the forces and torques generated from the thrusters, and the initial values of the rotational states. The first two items are either determined by an estimator, or are known constants. The forces and torques then must be determined from the thrust profile.

2.5.3 Determining How Thrusters Rotate and Translate SPHERES. The SPHERES plant does not directly interact with the controller to determine how to rotate and translate. Instead the plant rotates and translates based on the torques and forces that the thrusters induce onto the SPHERES system. In terms of the SPHERES simulation, the control signal must be interpreted the same as in reality. Therefore, the control algorithm output commands must be converted into a thrust vector. This thrust vector uses ones and zeros to represent which thruster is firing. The plant model must then interpret this signal to determine how the thrusters are moving the satellite. The thrust vector is a column vector containing twelve elements. The first element in the array refers to SPHERES thruster zero and the twelfth element in the vector refers to SPHERES thruster eleven[4]. Any value of one means that particular thruster is firing while any value of zero implies that specific thruster is off. The simulation uses a number of logical ('if') statements to determine whether or not the satellite is rotating and/or translating based on which thrusters are firing. Table 2.1 shows how the thrusters cause SPHERES to rotate and translate.

For example, if SPHERES fires thrusters numbered four and five the satellite translates along its 'Z' body-axis, where as if SPHERES fires thrusters four and eleven then the satellite would rotate about its 'X'-axis. As mentioned, the simulation uses a series of logical ('if') statements to determine the nominal rotation and translations in the body frame. Once each rotation and translation has been determined about

[4]The numbering scheme of 0-11 was retained to be consistent with the SPHERES hardware and C^{++} code.

Table 2.1: Thruster Effects in the Body Coordinate Frame [5]

Thr#	Nominal force direction			Nominal torque direction		
	x	y	z	x	y	z
0	1	0	0	0	1	0
1	1	0	0	0	-1	0
2	0	1	0	0	0	1
3	0	1	0	0	0	-1
4	0	0	1	1	0	0
5	0	0	1	-1	0	0
6	-1	0	0	0	-1	0
7	-1	0	0	0	1	0
8	0	-1	0	0	0	-1
9	0	-1	0	0	0	1
10	0	0	-1	-1	0	0
11	0	0	-1	1	0	0

each axis in the body frame, the values are then modified by a gain to scale them to correspond with an actual force or torque. The gain to adjust the force and torque signals is derived from the physical thruster location on SPHERES with respect to the center of mass. Figure 2.10 depicts the geometry of the thrusters causing a force and a torque on the satellite.

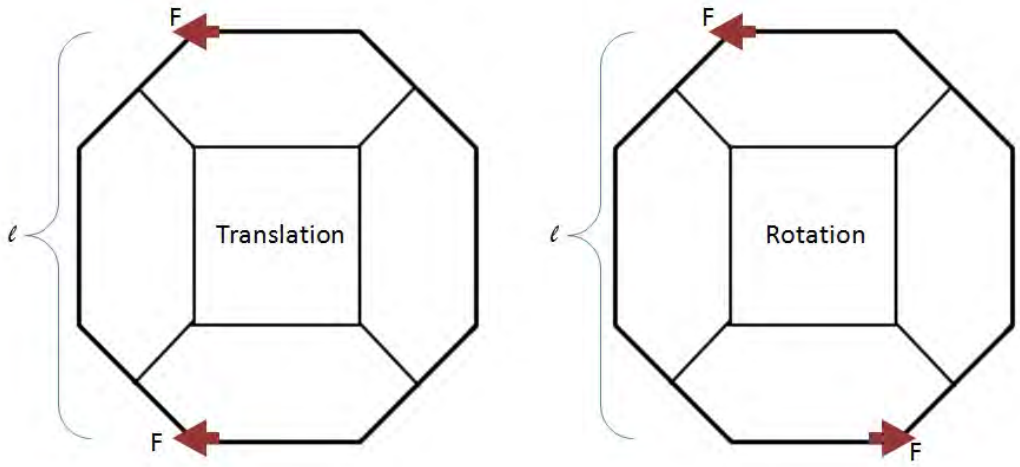

Figure 2.10: Thruster Location on SPHERES

Each force signal sent to the plant is the combination of two thrusters. Thus, each force signal is modified by '2F' where 'F' is the amount of force generated from one thruster. In addition, each torque comes from the moment generated by two thrusters. Therefore, assuming the center of mass is at the geometric center, the sum of these two moments generates a torque with the magnitude of 'Fℓ' where 'ℓ' is the diameter of the satellite, as shown in Figure 2.10[5].

Lastly, there are some hardware specific aspects of the plant that have yet to be described. Specifically, the periodic signal suppressor and saturation which are unique to this application need to be addressed. These nonlinearities are discussed with the controller nonlinearities of Section 3.2.6 because these nonlinearities as well as the other controller nonlinearities are directly applied to the control signals.

As the discussion of the SPHERES plant concludes, this research uses the techniques discussed within Chapter II and applies these concepts to create a speed and path control algorithm. Furthermore, the simulation used to operate the control algorithm is also developed and discussed in Chapter III.

[5]In reality SPHERES center of mass is 1.68 mm from the geometric center without fuel and 4.20 mm away when loaded with fuel [2]. This creates a small time dependent coupling of translation and rotation. These small effects are treated as disturbances.

III. Methodology

This chapter outlines the general procedure for designing a speed and path control algorithm for MIT's SPHERES program as well as implementing the algorithm into a simulation using the SIMULINK® program in MATLAB®. A basic model of this simulation can be seen in Figure 3.1. This figure shows four blocks that summarize how SPHERES is to be controlled. SPHERES itself, or the plant, contains all the information about the satellite responds when its thrusters are fired. The equations and nonlinearities of this system are described in Section 2.5 and are not covered in this chapter. The determine errors block determines the relevant state errors by processing information from the user commands and the current state vector. The controller then drives those state errors to zero by generating the thrust profile necessary to move the satellite as the user desires. This process is of course iterative as each force and torque generated from the thrusters changes SPHERES states which must be re-evaluated to ensure the satellite is where it is supposed to be.

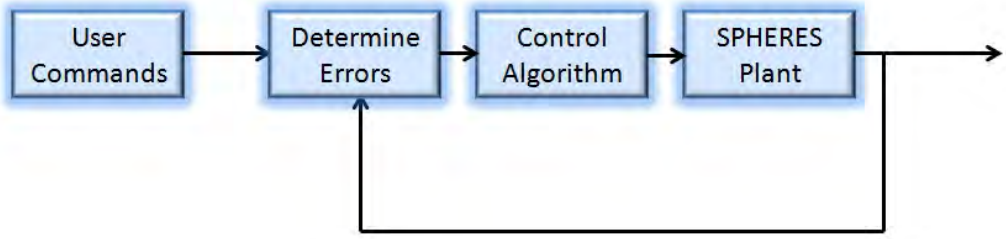

Figure 3.1: Simple Control Diagram for Simulation

Although Figure 3.1 illustrates the basic diagram for the SPHERES simulation developed in this thesis, SPHERES actually includes an estimator as well. Since the purpose of this thesis is to create a speed and path control algorithm for the SPHERES program, the simulation used to develop the controller does not include the estimator as it is assumed for this simulation that the estimator can provide the full-state feedback to the controller without any errors. While no estimator is truly

perfect, the SPHERES estimator currently in use is robust enough to generate the state values of the satellite for the applications of SPHERES [2]. With that in mind, the simulation described in this chapter follows the model presented in Figure 3.1.

This chapter is broken down into three sections. Section 3.1 covers how the simulation determines the state errors that the controller will need to drive to zero, and Section 3.2 describes the control algorithm block in Figure 3.1. In addition, Section 3.3 covers the User Commands block in Figure 3.1, and details what users would typically command and how those commands would be fed into the Determine Errors block.

3.1 Error Determination

This section discusses what is needed to determine the errors between the desired states and the current states of the satellite. While this is traditionally performed by finding the difference between the desired values and the current values of interest, this is not always the case for this application. Furthermore, when this method is appropriate, the desired values must then be conditioned before the errors are sent to the controller. The SIMULINK® simulation performs this task in the 'Error Determination' block, and the associated diagrams are included in Appendix B.1. Since the method to determine the position and velocity errors is different from the method to determine the quaternion error, each will be discussed separately.

3.1.1 Relative Errors. At this stage in the algorithm, the desired and actual values for the position and velocity of the SPHERES satellite are in the global frame. Thus, the difference of these vectors is used to determine the error between these vectors as shown in Equation 3.1.

$$\bar{X}_{error} = \bar{X}_{desired} - \bar{X}_{actual} \tag{3.1}$$

While this equation is straight-forward it is useful to note that the error formed by this equation still retains similar properties to the values used to create the error. To illustrate this, both the desired position and current position are vectors that represent where the satellite should be and where the satellite is located define by the body frame coordinate system. Thus, the position error is also a vector in the body frame that represents where the satellite needs to move to be in the desired position. The same is true for the velocity information. Lastly, if either the position or velocity error vectors are zero, then the satellite is in the correct position or velocity respectively.

Before the error can be input into the controller however, the error must be converted into the satellite's body frame because that is the coordinate frame used to operate the thrusters. This is completed by pre-multiplying the satellite's rotation matrix, R_{bi}, with the position and velocity errors. This matrix is determined using Equation 2.29 from Section 2.3.4 with the current quaternion values found in the satellite's state vector.

3.1.2 Pointing Error. In the general case, pointing accuracy is degraded through three different methods as shown in Figure 3.2. The knowledge error refers to errors between the estimate of how the satellite is oriented and how the controller believes the satellite is oriented. This error is minimized through the estimator. In regards to this simulation, the estimate and true orientations of the satellite are always the same since full-state feedback is used without any noise. Since the knowledge error is dependent on the estimator, this error is ignored for this research. The control error is the difference between where the controller thinks the satellite is pointing and where the satellite should be pointing. As the name implies, this error is directly related to the performance of the control algorithm. Thus, reducing this error is the focus of the quaternion controller. Since this simulation has no knowledge error, the control error is synonymous with the satellite pointing accuracy. The stability of the true vector is the concern of the third error.

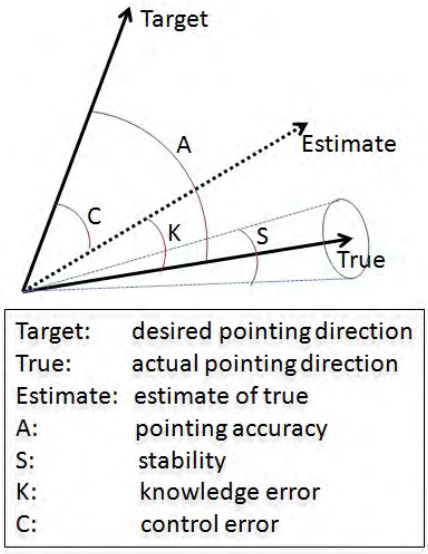

Figure 3.2: Sources for Orientation Error

Although controllers and estimators can command the true vector to line up with the target vector, the true vector is not perfectly parallel with the target vector. In fact, the true vector will move around the target vector in a small cone as the true vector is being adjusted to make the two vectors parallel. This results in the true vector appearing to 'jitter' about the target ('S' in Figure 3.2). This jitter may or may not be an issue though. Sensitive sensors demand that the jitter be corrected. However, this is not an issue if the field-of-view is larger than the error generated from jitter, and if the sensor has a high enough frame rate to ensure the motion from jitter does not cause a blurring effect. Since this error is payload dependent it is usually corrected with a specific payload controller that damps out any vibrations or other disturbances produced by the satellite. Since a specific sensor package has not been suggested for this controller the author chose to confine the allowable jitter to a two degree field of view. This restriction allows most sensor applications to be applied to SPHERES that are capable of being installed on SPHERES without wasting an excessive amount of fuel to correct the remaining jitter can be handled by the sensor. In any case, the pointing error that needs to be fixed is the control error ('C' in Figure 3.2). This is corrected by determining the quaternion error.

3.1.3 Quaternion Error. While calculating the relative errors are relatively intuitive before the coordinate transformations, the method to determine the quaternion error is a little more abstract. This is because the quaternion error is itself a quaternion vector that describes how to rotate from the satellites' current position to the desired position. Recall that quaternions, by design, always have a magnitude of one to avoid singularities. Since the quaternion error is still a quaternion in and of itself, the quaternion error must have a magnitude of one as well. Therefore the simple error calculation found in Equation 3.1 does not work for quaternions as the difference between the desired quaternions and the current quaternions would not necessarily result in a quaternion vector that described how the satellite should rotate to get into the desired orientation. Thus, a separate equation is used to determine the quaternion error, $\bar{q}_{error}$.

The quaternion error is traditionally found by using the desired quaternions to populate the transmuted quaternion matrix discussed in Section 2.3.5. In this application however, that information is not provided. Instead, three vectors are available. Both position vectors of the object and the satellite are provided in the global frame. In addition, the user has supplied a pointing vector in the body frame. This pointing vector specifies what part of the satellite is supposed to face the object of interest (typically the target). The quaternion error should describe how to rotate the satellite so that its pointing vector is facing the target. To do this two vectors are needed in the body frame: the target vector and the pointing vector. The target vector is generated by taking the unit vector that results from the difference between the position vector of the object the satellite is suppose to face and the position vector of the satellite. This difference is calculated in the global frame and converted to the body frame. The cross product of the target and pointing vectors produces a vector that is normal to the two vectors, which can be used as the eigenaxis of rotation. The principal Euler angle is then the angle between the two vectors. Equation 3.2 shows how to determine the principal Euler angle from the target vector, $\bar{T}$, and the pointing vector, $\bar{P}$.

$$\Phi = acos(\frac{\bar{T} \cdot \bar{P}}{|| \bar{T} || || \bar{P} ||}) \tag{3.2}$$

The eigenaxis of rotation and principal Euler angle are then converted into quaternions. This is accomplished using the equations presented in Section 2.3.4. Recall, that the eigenaxis method of rotation produces a singularity when no rotation is required. This occurs because the vector for the eigenaxis of rotation could point in any direction. This singularity is avoided through the use of a cross product however. This is because no rotation would mean that the target vector and the pointing vector are parallel, and the cross product of two parallel vectors results in a zero vector. Thus, the quaternion vector formed from this eigenaxis and principal Euler angle always accurately reflect the required rotation to maneuver from the current orientation to the desired orientation. Once the quaternion error is computed and the translational errors have been rotated into the body frame, the errors are sent to the speed and path control algorithm to minimize the errors in the satellite's position, velocity and orientation.

3.2 Control Algorithm Development

With an understanding of the state errors, the Speed & Path Controller compartment in Figure 3.1 now has the necessary information to generate the control force and control torque signals. This section discusses the speed and path control algorithm as a whole and then briefly discusses how each sub-section is broken down. The speed & path control algorithm discussed herein commands SPHERES' location and orientation by using two independent controllers. One controller modifies the translational states by using a look ahead controller with optimal gains determined by a Linear Quadratic Regulator to control the position and velocity of the satellite's location. The other controller drives the first three quaternion errors to zero by producing a stable control law based on a Lyapunov equation. Figure 3.3 displays a diagram of how these controllers appear in the simulation.

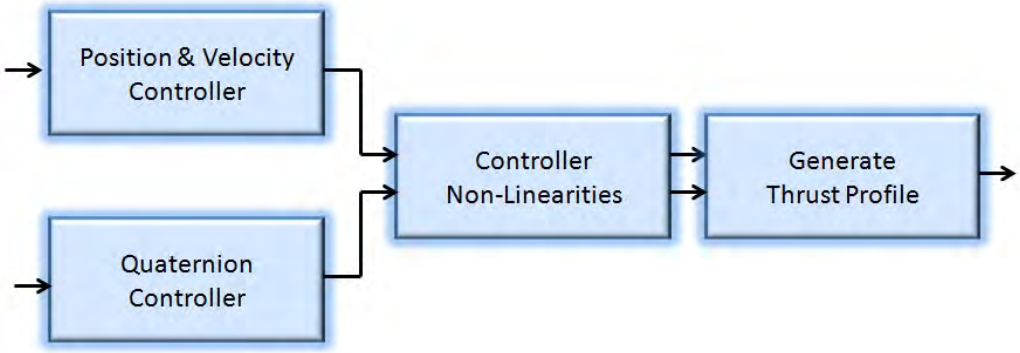

Figure 3.3: Diagram of Position, Velocity, and Quaternion Controllers

The control algorithm encompasses four major components which each accomplish a separate objective. The procedure for the translational controller is discussed in Section 3.2.1, while selecting the optimal values for this controller is covered in Section 3.2.2. The methodology for the quaternion controller is explained is Section 3.2.4, and the selection of this controller's optimal gain value is covered in Section 3.2.5. In addition to the discussion of controllers, the controller nonlinearities are included in Section 3.2.6, and the logic to generate the thrust profile is conveyed is Section 3.2.7.

3.2.1 Translational Position & Velocity Controller. The control algorithm commands the satellite's position and velocity in the body frame by using a bang-bang controller with weights determined through an LQR. The bang-bang controller design is used partly because this represents what the thrusters physically are, and partly because this technique allows one to incorporate the velocity control relatively simply through the use of a 'look-ahead' gain, τ. Details of this type of controller can be found in Section 2.4.4. Adjusting τ changes the slope of the switch line for the control algorithm as described in Equation 3.3.

$$M_{switchline} = \frac{K_{position}}{K_{velocity}\tau} \tag{3.3}$$

Equation 3.3 shows that an inverse relationship exists between slope of the switch line, $M_{switchline}$, and τ. The additional two terms are gains that weight position and velocity[1]. Recall that adjusting the switch line shifts the importance of correcting position versus velocity. In essence, τ provides users with a 'knob' to adjust which parameter is more desirable to correct: position or velocity.

In addition to the bang-bang controller an LQR is implemented into the control algorithm to make the design more robust. Recall from Section 2.4.5 that the LQR provides guarantees of the gain and phase margin of the system. This is accomplished through weighting position, velocity, and control. Figure 3.4 shows a sample case of how an LQR is used to control the position and velocity error of SPHERES.

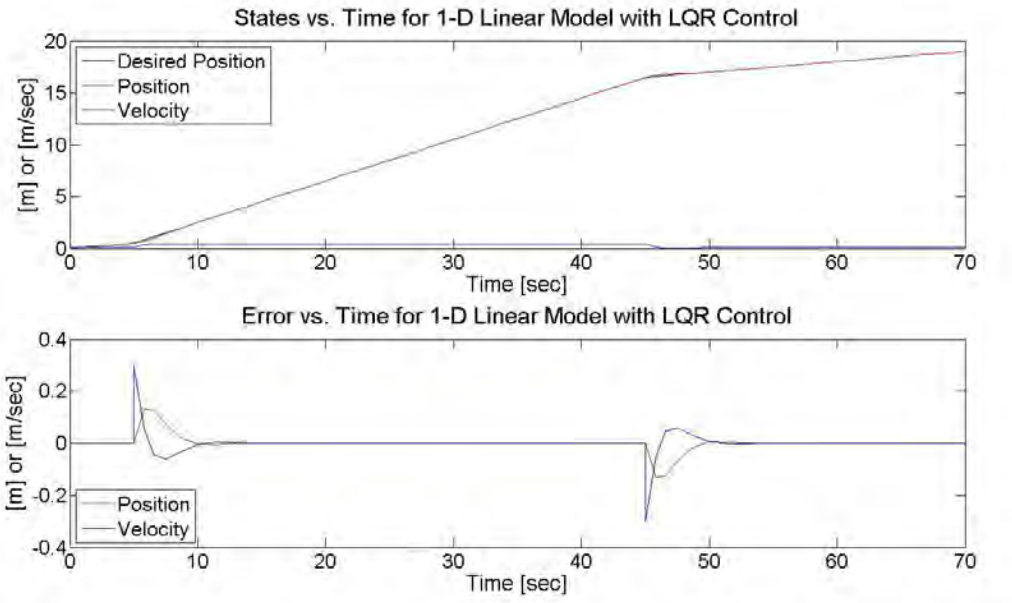

Figure 3.4: Translational Controller without Nonlinearities

The example in Figure 3.4 only considers one dimension for translating SPHERES, and does not include system nonlinearities. The simulation begins with SPHERES at the origin with an initial velocity of $0.1\frac{m}{s}$. At five seconds SPHERES is commanded to travel at $0.3\frac{m}{s}$ for forty seconds before slowing back down to the initial speed.

[1]These terms are the optimal gains derived from the LQR discussed later, and are included only to provide an accurate description of $M_{switchline}$.

Figure 3.4 shows the best-case response characteristics that the nonlinear controller can achieve since introducing the nonlinearities inherent within SPHERES can only degrade system performance. In addition to the time response of the system, the phase and gain margins of this example can be seen in the Nyquist plot of Figure 3.5.

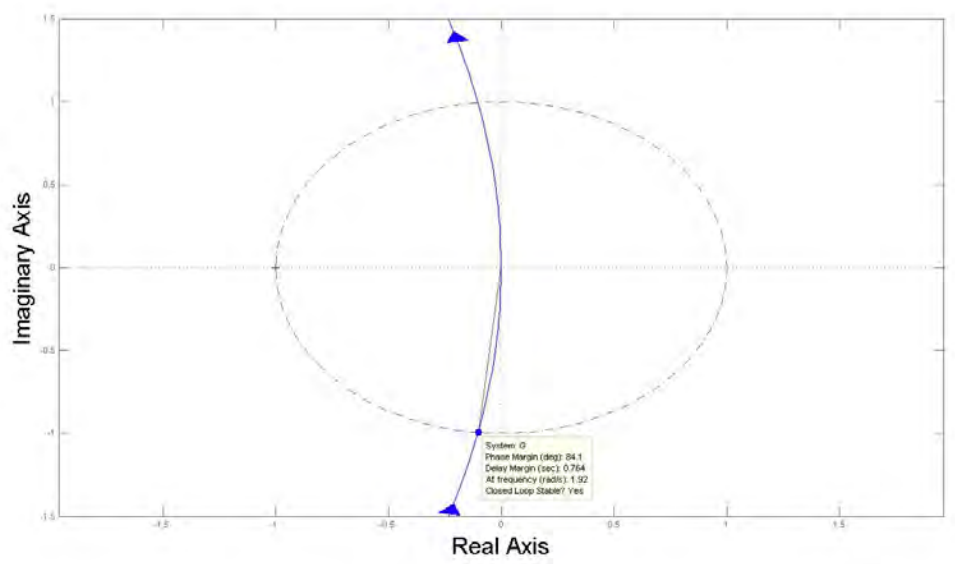

Figure 3.5: Nyquist Plot of LQR Controller without Nonlinearities

Figure 3.5 verifies that the LQR controller does in fact meet the minimum guarantees as the phase margin is greater than sixty-five degrees and the system has infinite gain margin. This is important because Nyquist techniques for stability analysis cannot be applied to nonlinear systems. These margins provide the best case scenario that any controller could mimic with nonlinearities present. Thus, the robustness inherent within these margins indicate that the a comparable nonlinear system could still possess some level of robustness when noise is present. The next step is to compare this sample response (Figure 3.4) with a more realistic model. To do this the controller will need to include the nonlinearities discussed in Section 3.2.6. As expected, once the nonlinearities are introduced however, the controller performance is degraded.

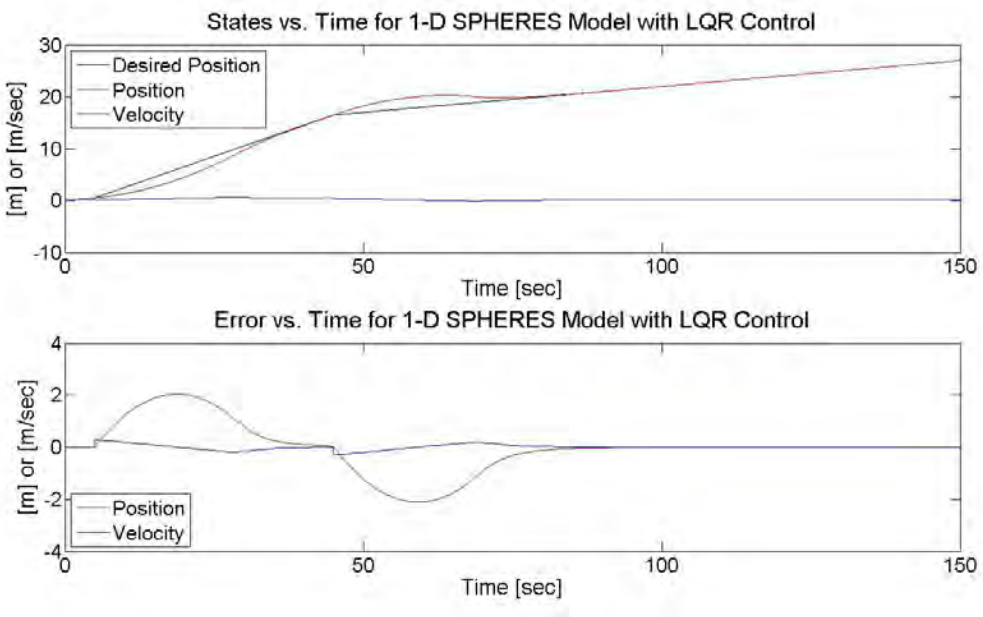

Figure 3.6: Translational Controller with Nonlinearities

Figure 3.6 displays system performance when the non-linearities discussed in Section 3.2.6. The largest contributor to degrading system performance is from the periodic signal suppressor (discussed in Section 3.2.6.4) because this limits the amount of time the controller can run. Since the nonlinear controller is only running 40% of the time that the linear controller runs, the nonlinear controller requires more time to fix the same amount of error. Although the system is stable, the transient response of the system are not particularly stellar. The transient response characteristics of the translational controller can be adjusted by changing the slope of the switch line and modifying the state and control weighting matrices with the LQR component of the controller. Section 3.2.2 discusses how the optimal values are determined for the LQR matrices, and Section 3.2.3 discusses how to best optimize τ to determine the best slope for the switch line for system performance. This is done in reverse order because the ratio from the LQR gains also affect the slope of the switch line as seen in Equation 3.3.

3.2.2 Optimal Weighting for LQR. As previously mentioned, the LQR algorithm is a prescriptive method to optimize the controller to minimize the cost function, but the particular weights on each of the states ($\mathbf{Q}$) as well as the control ($\mathbf{R}$) still need to be determined to ensure the controller meets design goals and stays within control limitations. In order to determine the $\mathbf{Q}$ and $\mathbf{R}$ weighting matrices for the LQR algorithm, a MATLAB® simulation was developed to take the current one-dimensional position and velocity controller and vary the weights for the states associated with position (Q_1) and velocity (Q_2) as well as the weighting for control (R_1). These weights form the $\mathbf{Q}$ and $\mathbf{R}$ matrices as shown below.

$$\mathbf{Q} = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \tag{3.4}$$

$$\mathbf{R} = \begin{bmatrix} R_1 \end{bmatrix} \tag{3.5}$$

It is useful to identify the performance parameters that will allow the controller to be shaped to desired design goals. Recall that the SPHERES program requires the ability to handle multiple changes in velocity as a satellite moves along the desired path. This desire is harder to achieve when the position and velocity errors are not corrected in a timely manner. Since the LQR algorithm is being applied to reduce the position and translational velocity errors, the transient response characteristics such as settling time and percent overshoot become the driving performance parameters to select the $\mathbf{Q}$ and $\mathbf{R}$ weights. Figure 3.7 below shows a system response of the errors in translational position and velocity in a simulated one-dimensional case.

In this sample case, a desired velocity change of 0.2 meters per second was commanded one second into the simulation. This can be easily seen in the velocity error as there is a sudden error of 0.2 at one second. In addition, the position error is almost a meter off before the controller compensates for the overshoot. Furthermore, both the position and velocity errors take over thirty seconds to be within 2% error.

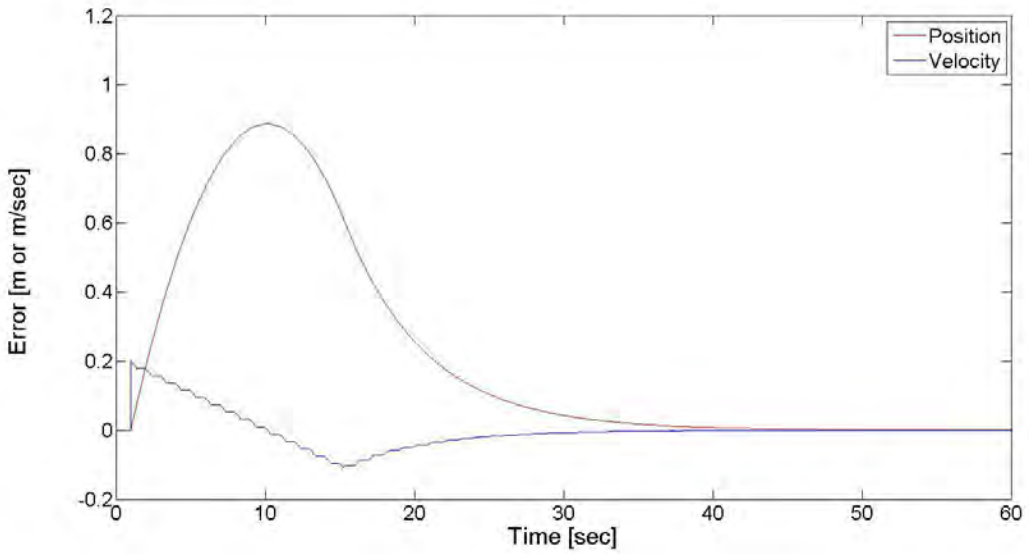

Figure 3.7: Sample Response for Position and Velocity Errors

Now it is important to note that the SPHERES are capable of a max change in velocity of 0.01 meters per second in a 0.4 second interval. Therefore the commanded input to instantly change velocity by 0.2 meters per second would be a bit of an unreasonable command if these changes weisat distinction is of importance to users creating paths for SPHERES to operate on, as the SPHERES acceleration is limited and should be taken into account for path and speed planning[2]. In regards to finding the optimal weights for the **Q** and **R** matrices, the actual values for the percent overshoot and settling time of the translational errors are not important so much as how the values compare to the other values in a test run. As such, the values for percent overshoot and settling time are normalized to allow for ease of analysis and comparison on plots. Since each weight placed on the optimal controller is only relevant when considered with the other weights placed on the controller, the ratios between these weights are of more interest than the individual weights themselves. Thus, while the position weight, the velocity weight, or the control weight could each be adjusted, only the ratio of the weights to each other affect the LQR controller. In addition only two

[2]If SPHERES is subject to user imposed rate limits (discussed in Section 3.2.6.3) then those limits must also be considered for speed and path planning.

independent ratios are of any use as any other ratio derived from three variables is redundant. Therefore, to begin the test the user has the option to manipulate two independent ratios or two knobs to adjust and determine the optimal weights for **Q** and **R**. For the first test, the author chose to vary the weighting ratio between the control and position weights or R_1/Q_1, and for the second test the author chose to use the ratio between the velocity weight and the position weight or Q_2/Q_1. Since each test only changed one of the two available ratios or knobs, the other knob was left stationary. Since no information of an optimal ratio was available for the first test the second ratio of Q_2/Q_1 was set to a value of one. For the second test, the ratio of R_1/Q_1 was set to a constant value equal to the optimal value derived from the first test. Figure 3.8 shows the first test run. Each case considered one thousand linearly spaced ratios between the values of 0.0014 and 1.4 because this range contained all the tradeoffs that were to be considered based on the performance parameters.

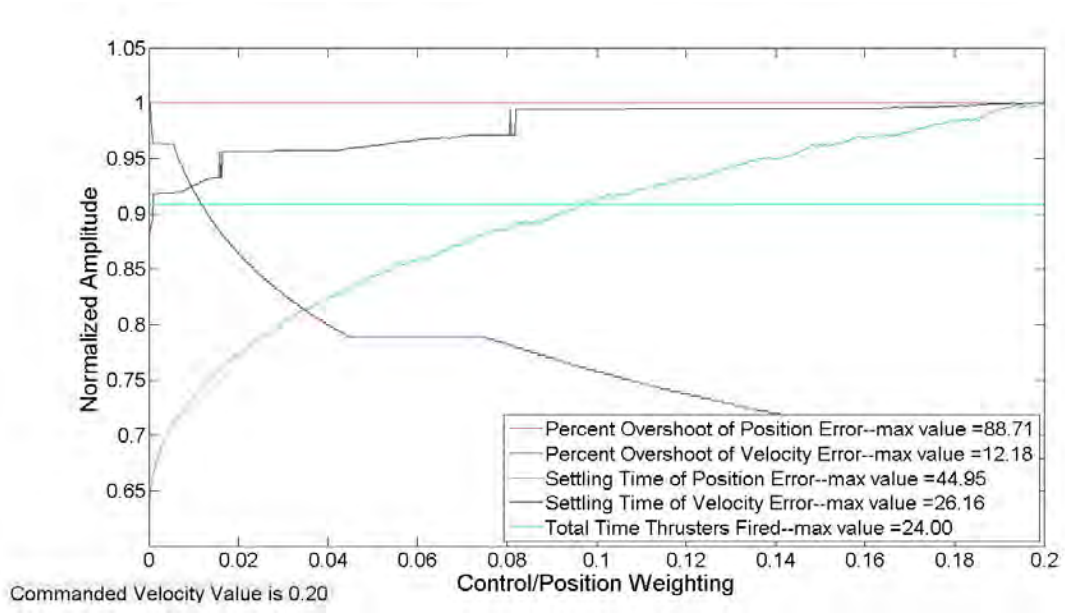

Figure 3.8: Performance Characteristics as R_1/Q_1 Changes & Input is 0.2 m/s

This plot shows that as the ratio of the control weight over the position weight increases, the percent overshoot of the velocity error decreases while the settling time of both of the errors increases. The percent overshoot of the position error

and the amount of time the thrusters fired both remain constant and do not change with respect to the weighting ratio tested. Additionally, the max values of each performance parameter located in the legend allows one to compare how significant each parameter is under these conditions. For this test case only three parameters were changed, the percent overshoot of the velocity error as well as the two settling times. In addition, there is a tradeoff between fixing the percent overshoot versus the two settling times because as the ratio of the control weight to the position weight increases, the percent overshoot of the velocity error drops and the two settling times increase. The opposite is true when the weighting ratio is lowered. Since a percent overshoot of 11% is not going to prevent the SPHERES from performing well, and the settling times of thirty and sixty seconds could, the author chose to fix the settling times. Therefore this plot would suggest that the best weighting ratio would be for the control ratio to be really low compared to the position error weight. But this test was run when the commanded velocity was 0.2 meters per second which the author is assuming would be the highest instant change in translational velocity that one would command for SPHERES in non-typical circumstances (non-typical because the SPHERES physically takes about a minute to catch up with the command, which is one-sixth of the normal SPHERES test runs). How do the system response characteristics affected when the commanded velocity changes are in a typical operating range? Figure 3.9 and Figure 3.10 show how the system changes for smaller inputs.

The most noticeable difference in these plots is the change in the settling time of the velocity error. While commanded velocity inputs of 0.2 and 0.05 meters per second (Figure 3.8 and Figure 3.10 respectively) indicate the ratio of the control weight to the position error weight should be as low as possible to minimize the settling time of the velocity error, the settling time of the velocity error actually increases as the weighting ratio decreases below 0.4 when a commanded velocity value is 0.1 meters per second as seen in Figure 3.9. The other noticeable difference is in the percent overshoot of the velocity error. Although this system parameter continues to display an indirect

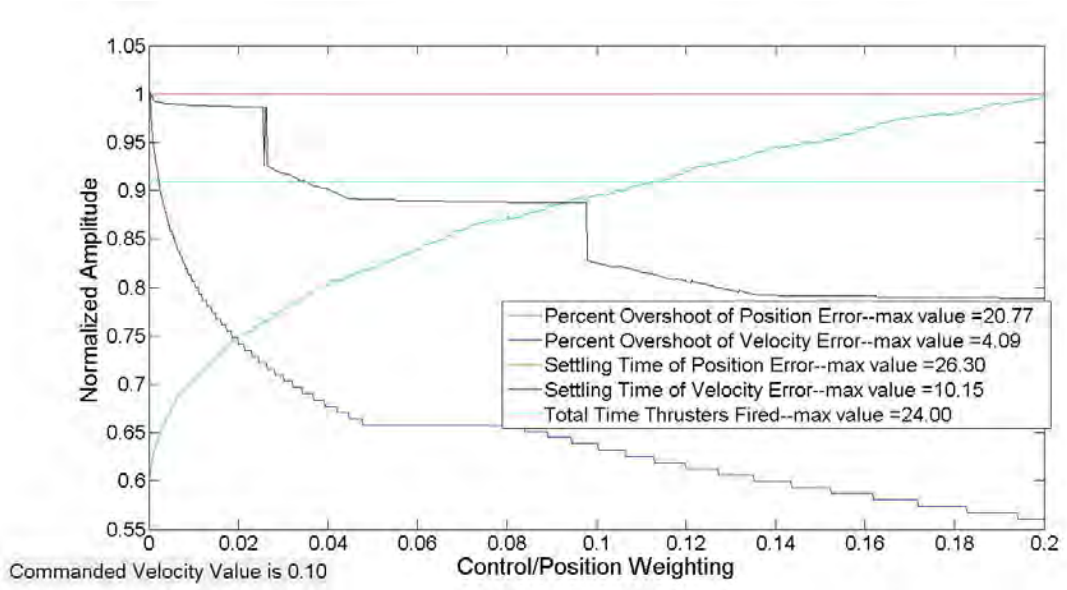

Figure 3.9: Performance Characteristics as R_1/Q_1 Changes & Input is 0.1 m/s

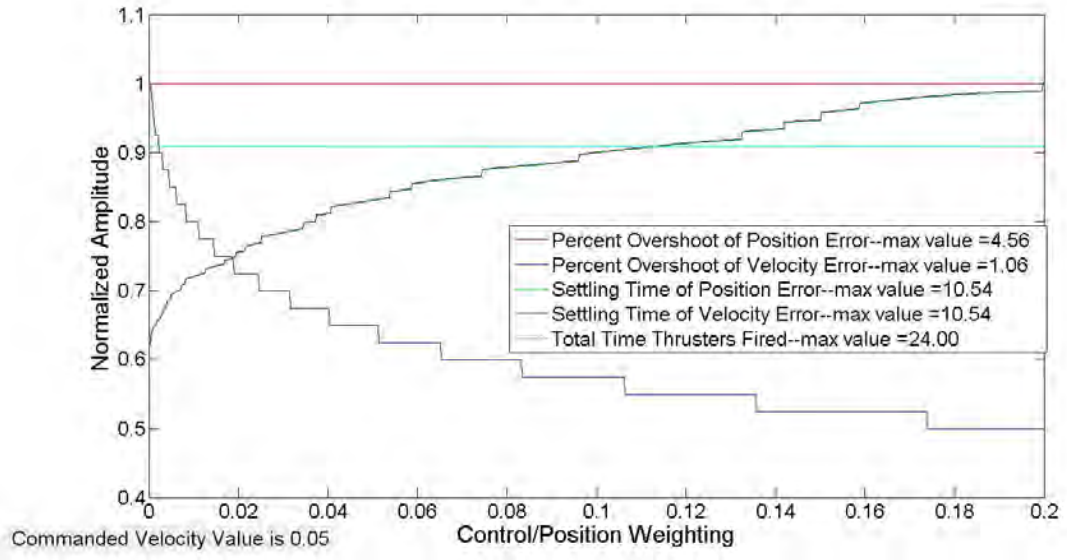

Figure 3.10: Performance Characteristics as R_1/Q_1 Changes & Input is 0.05 m/s

relationship with the weighting ratio as the commanded change in velocity changes value, the shape at which this parameter follows exhibits a larger drop when the ratio is below 0.2. These two differences indicate that the control to position weighting ratio should not be as low as possible as in the case for when the commanded change in velocity is 0.2 meters per second. In addition, with the case 1 when the change in velocity was 0.2 meters per second, the author placed more emphasis toward fixing the settling times than the percent overshoot of the response of the errors as the settling times of the response continue to have a larger impact on the overall function of SPHERES than the percent overshoot does. Furthermore, when the commanded change in velocity is 0.05 meters per second and lower the actual values of the system parameters considered are all within acceptable regions regardless of what the weight ratio is. Therefore the difference between the control and position error weights for the LQR controller has the most impact with the desired change in velocity is between 0.05 and 0.2 meters per second as in Figure 3.9. In these instances the author chose to prioritize the settling time of the velocity error over the settling time of the position error since this control algorithm is primarily tasked to control the velocity error. Therefore the author chose to select the ratio of the control weight to the weight of the position error to be 0.3.

With the ratio between the control and position error weights determined, the next step was to pin down an acceptable ratio between the weights of the velocity and position errors. For these simulations the ratio between the weight of control and position error was preserved at 0.3. The procedure for determining the ratio between the weight for the velocity error (Q_2) and the weight for the position error (Q_1) is similar to that of finding the ratio of the weights for control and position error. Thus for brevity the following three plots show how the same system parameters change as the ratio of Q_2/Q_1 increases. Figure 3.11 shows the changes in system parameters when the commanded change in velocity is 0.2 meters per second, while Figure 3.12 and Figure 3.13 respectively display how the system parameters change when the change in velocity is commanded to be 0.1 and 0.05 meters per second.

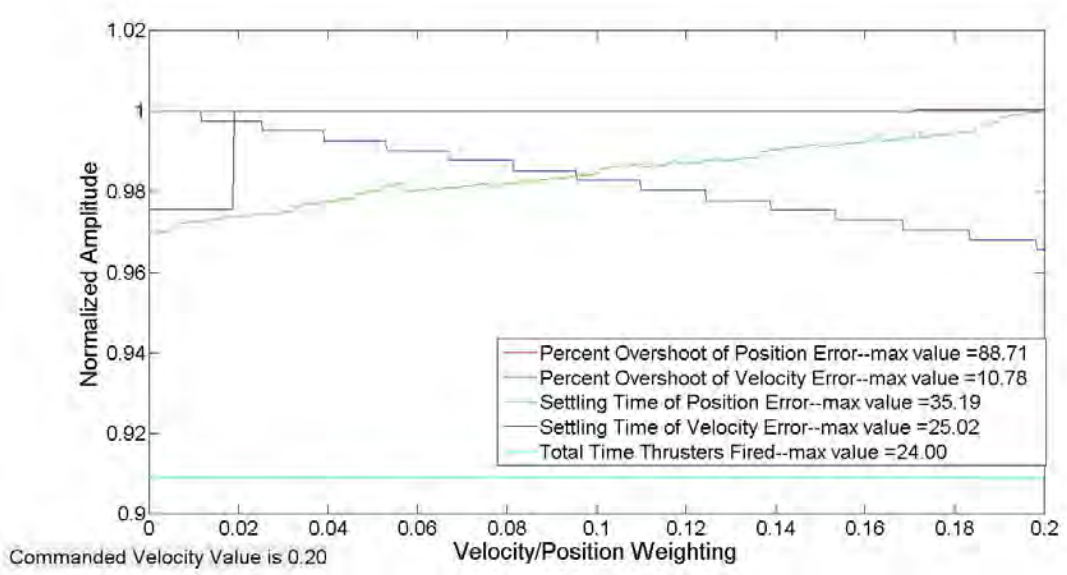

Figure 3.11: Performance Characteristics as Q_2/Q_1 Changes & Input is 0.2 m/s

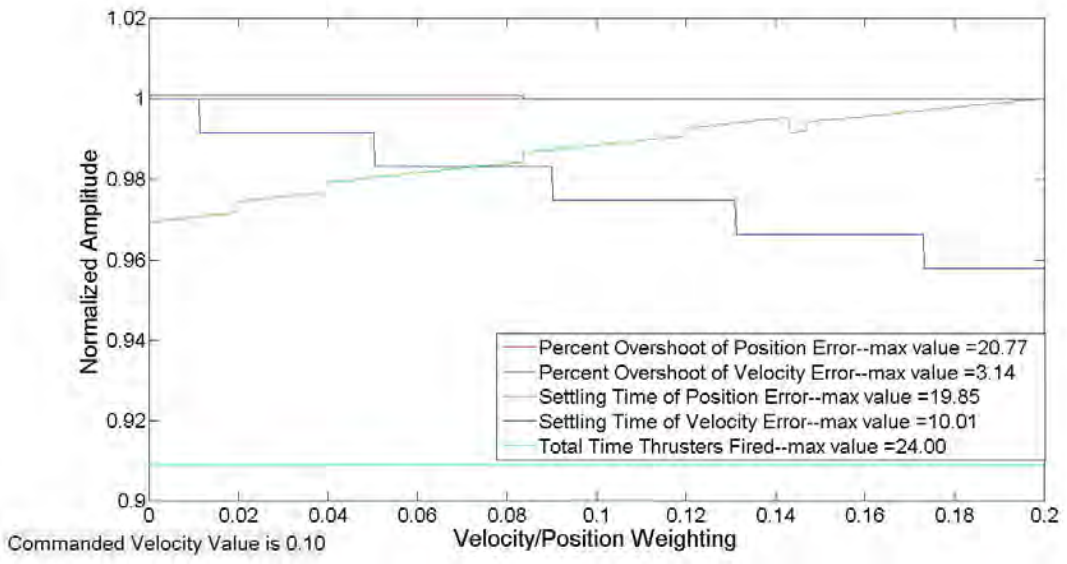

Figure 3.12: Performance Characteristics as Q_2/Q_1 Changes & Input is 0.1 m/s

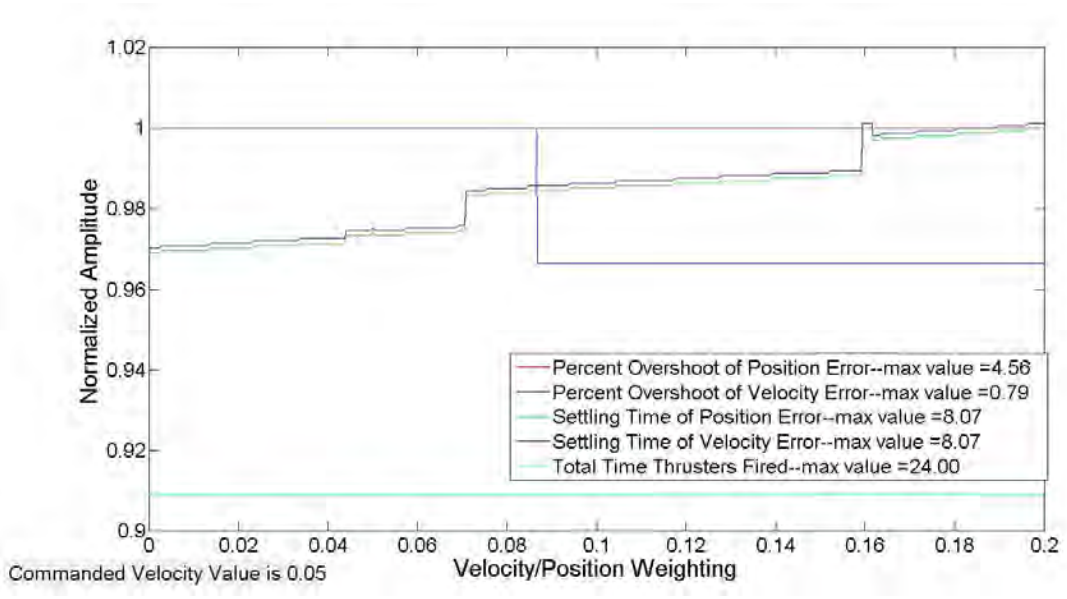

Figure 3.13: Performance Characteristics as Q_2/Q_1 Changes & Input is 0.05 m/s

In these simulations the percent overshoot of the position error is not affected by a change in the ratio of Q_2 to Q_1 while the percent overshoot of the velocity error is negatively correlated to the same ratio. Furthermore, the settling time for the position error is positively correlated to the ratio of Q_2 to Q_1. The settling time for the velocity error behaves differently however. Although the settling time for the velocity error is positively correlated in Figure 3.11 and Figure 3.13, this parameter is negatively correlated in Figure 3.12. Lastly, the total time the thrusters were firing was constant regardless of the commanded velocity inputs or the ratio of the weights Q_2 and Q_1. This is attributed to two reasons. First, the change in weights for this part of the trade study has no affect on the change in the control weight therefore the same weighting on control is applied to all the simulations therefore resulting in the same control usage as one varies the ratio of Q_2 to Q_1. Secondly, the fuel time is the same regardless of the input because when the commanded input is relatively high for the system the controller is using the thrusters to drive the errors to zero while the input is relatively low for satellites, the controller is expelling fuel to maintain the errors. One might suggest expanding the region to apply the dead zone to minimize the fuel consumed but doing so degrades the accuracy of the position and velocity

of SPHERES. For missions like in-space robotic assembly and other close proximity operations between satellites with multiple maneuvers in which satellite refueling is an option, maintaining accuracy is of utmost importance. Therefore the control usage (as seen through the length of time the thrusters fired) is observed to insure the control reach unnecessary levels, but otherwise ignored.

Once again, the performance parameters of interest are the percent overshoot of the velocity error and the settling times of both the position and velocity error. When the commanded change in velocity is lower than 0.05 meters per second the system performance parameters are all in acceptable regions regardless of what the weights of velocity and position errors are. Therefore for the purposes of selecting the weighting ratio of Q_2 to Q_1, these cases can be ignored. Furthermore the percent overshoot of the response of the velocity error does not exceed 11% thus the real parameters of interest are the two settling times. When the value of the commanded change in velocity is relatively high as in Figure 3.11, the settling times are both positively correlated, so an extremely low ratio for the weights of Q_2 to Q_1 is desirable. Yet in the middle range of operation as in Figure 3.12 the settling time for the velocity error is now negatively correlated. This difference yields two values for the ratio of Q_2 to Q_1 that would be acceptable. They are when the ratio is 0.09 and 0.37. Table 3.1 displays the system parameters of interest that each weighting ratio would produce under the given circumstances.

Table 3.1: Comparison of System Parameters when Q_2/Q_1 is 0.09 and 0.37

Commanded Velocity	Ratio of Q_2 to Q_1	Percent Overshoot of Velocity Error	Settling Time of Position Error	Settling Time of Velocity Error
0.2	0.37	9.64%	37.26 sec	25.05 sec
0.2	0.09	10.75%	34.26 sec	24.40 sec
0.1	0.37	2.75%	21.23 sec	9.07 sec
0.1	0.09	3.12%	19.26 sec	10.01 sec

As shown in Table 3.1 the settling times exhibit an indirect relationship in that lowering one raises the other and vice versa. The author chose to select the ratio of

Q_2 to Q_1 to be 0.09 for two reasons. First, while the control algorithm is designed to control the velocity of SPHERES, the settling time of the position error is over thirty percent larger than the settling time of the velocity error. And the position error has undesirable values for the settling time while the velocity value has at least acceptable values. Lastly, while the selection of either ratio value would make one of the settling times worse, the selection of 0.09 as a value for the ratio of Q_2 to Q_1 has less of a harmful affect on the settling time of the velocity error than the other ratio value has on the settling time of the position error. With the ratio between both R_1 to Q_1 and Q_2 to Q_1 set, the selection of any arbitrary value of Q_1 would automatically set the values of R_1 and Q_2 to properly weight the LQR controller to deliver optimal results. Before moving on, the author chose to perform another test to verify an assumption that was made at the beginning of this trade study. Earlier an assertion was made stating only two ratios are needed to know how to set the LQR controller to generate the best performance of the system even though three weights are required. If this is true one would expect that changing the ratio of Q_2 to R_1 would not affect the system parameters of the response in either the position and velocity errors. In this last test the ratio of the weight of the velocity error to the weight of the control is changed at different inputs to see how the system parameters change. This test shows that varying the velocity error weight with respect to the control weight did not affect the system in any significant ways. Only changes of less than a percent occur in any of the measured system parameters. Therefore it is valid for one to assume that for this system only knowledge of the two ratios were needed to determine the optimal weights for the LQR controller.

Equations 3.6 and 3.7 show the final weighted matrices derived for the LQR controller. These matrices are created by arbitrarily selecting the weight for the position error and picking the other weights so that the optimal ratios are preserved.

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 0.018 \end{bmatrix} \tag{3.6}$$

$$\mathbf{R} = \begin{bmatrix} 0.06 \end{bmatrix} \tag{3.7}$$

3.2.3 Optimal Weighting for τ. With the state and control weight matrices selected, the feedback gains for the position and velocity components are determined following the process described in Section 2.4.5. Once the feedback gains are determined the switch line of the bang-bang controller is only dependent on τ. Figure 3.14 illustrates how adjusting τ changes the slope of the switch line.

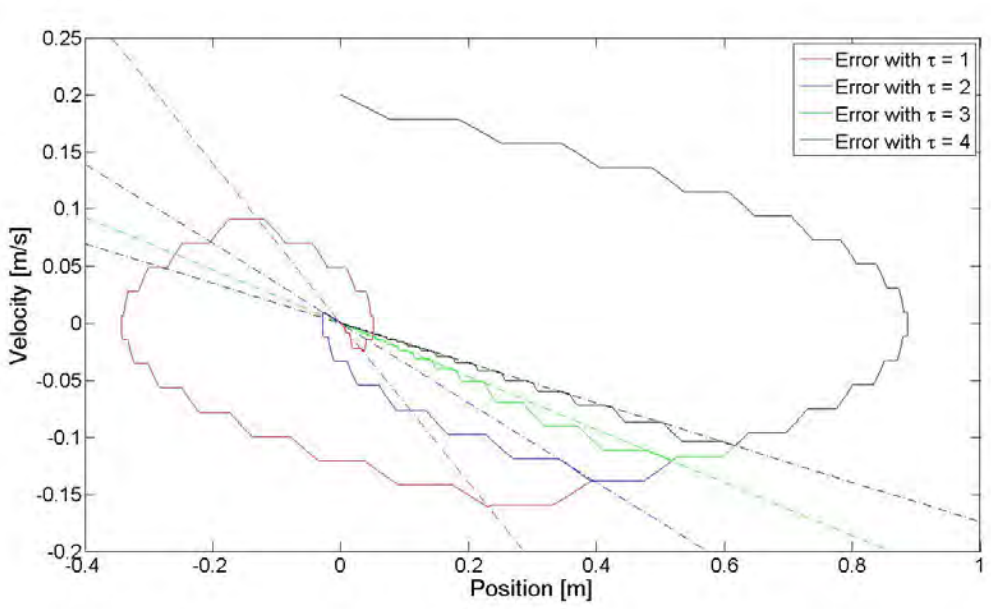

Figure 3.14: Phase Plane of Translational Errors for 1-D Simulation

Figure 3.14 also displays a how the relationship between the position and velocity errors[3] changes with τ. When τ is too big or too small, the slope of the switch line becomes too shallow or too steep respectively. This in turn distorts the relationship of the errors by skewing priority between which error should be minimized at a specific point in the phase plane. Furthermore, changing τ also changes the time response

[3]Recall that the controller cannot run simultaneously with the SPHERES estimator. Thus, when the controller is off, the velocity is temporarily 'stuck' while the position error continues to drift before the controller is turned on again. This phenomena results in the horizontal lines occurring throughout the relationship of the errors.

of the SPHERES model. Figure 3.15 shows the time response of the position and velocity errors as τ is adjusted from one to four.

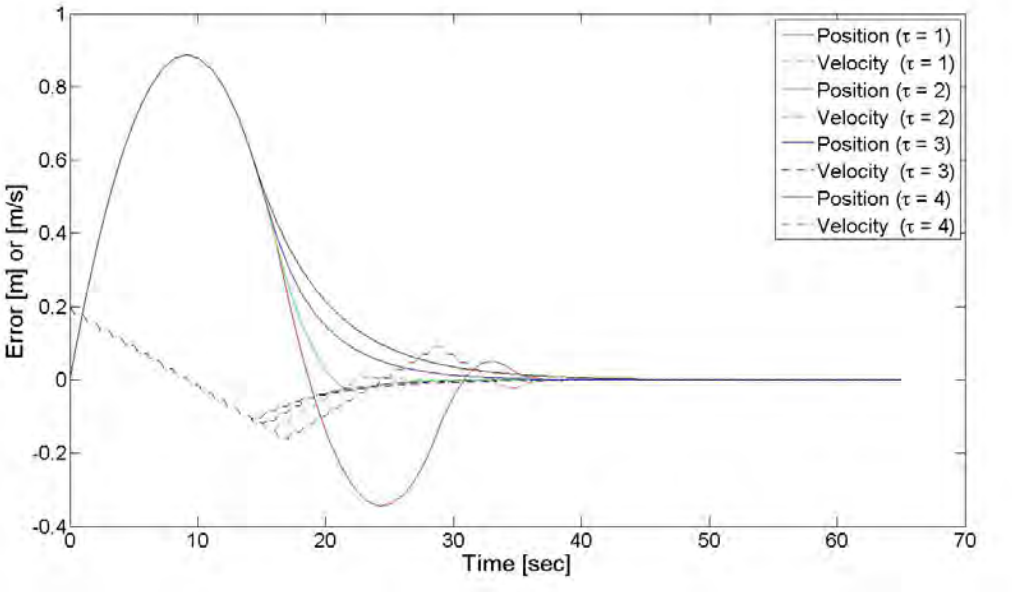

Figure 3.15: Response of Translational Errors for 1-D Simulation

Notice in Figure 3.15 how all the position or velocity errors begin in the same manner but diverge one at a time as τ changes. This is because the simulation always begins above the switch line regardless of the given values of τ, and each diverging response occurs as phase plane trajectories pass their corresponding switch line. In addition, note that the extreme values of τ produce undesirable time response plots most noticeably in regards to the settling time of the response. This indicates that some optimal value of τ exists to produce desirable response characteristics.

In order to determine a desirable value of τ, the author chose to consider the system's rise time and settling time as performance characteristics the desired performance characteristics. This is because changing τ could produce under-damped or over-damped responses. The settling time (2% method) is used to rate how fast each response reaches the final value while the rise time (10% to 90%) is used to indicate how quickly the desired response gets to the desired region. While the rise time may appear to be a redundant comparison, this performance parameter is included

because of the controllers applications. New user inputs could be fed into the control algorithm quicker than the controller minimizes the errors of the previous command. Thus it is important to reduce most of the error quickly to minimize the effect of compounding previous errors with new errors. Furthermore, the rise-time is actually the most important consideration for this reason. In addtion, the percent overshoot was not considered for this application because under-damped systems with relatively quick rise-times have minimal values for percent over-shoot, so no beneficial information is gained by including this value. Lastly, only the position error was considered for much of the same reasons as no additional information exists within the velocity errors when the position error is already considered. Table 3.2 contains a brief summary of the results.

Table 3.2: Comparison of System Parameters of Position Error as τ Changes

τ	Settling Time [sec]	Rise Time [sec]	Response
1.0	25.85	5.94	under-damped
1.5	20.90	6.03	under-damped
2.0	14.52	6.92	under-damped
2.5	13.25	7.35	over-damped
3.0	19.54	10.03	over-damped

Table 3.2 indicates the best performance results when $\tau = 2$. Values of τ in between those listed are not included to keep the table manageable. Furthermore, values of τ between 2.0 and 2.5 do not provide significant differences. This is because the response becomes critically damped somewhere between $2.4 < \tau < 2.5$ and as this happens it is almost an instantaneous switch. Thus $\tau = 2.0$ is in fact the most desirable value.

Now that all the gains for the translational controller have been determined Figure 3.16 & Figure 3.17 display a sample response of this controller in one dimension.

Since each dimension is independent, the three dimensional control algorithm is simply the one-dimensional controller applied to the three different axes in the satellite's body frame.

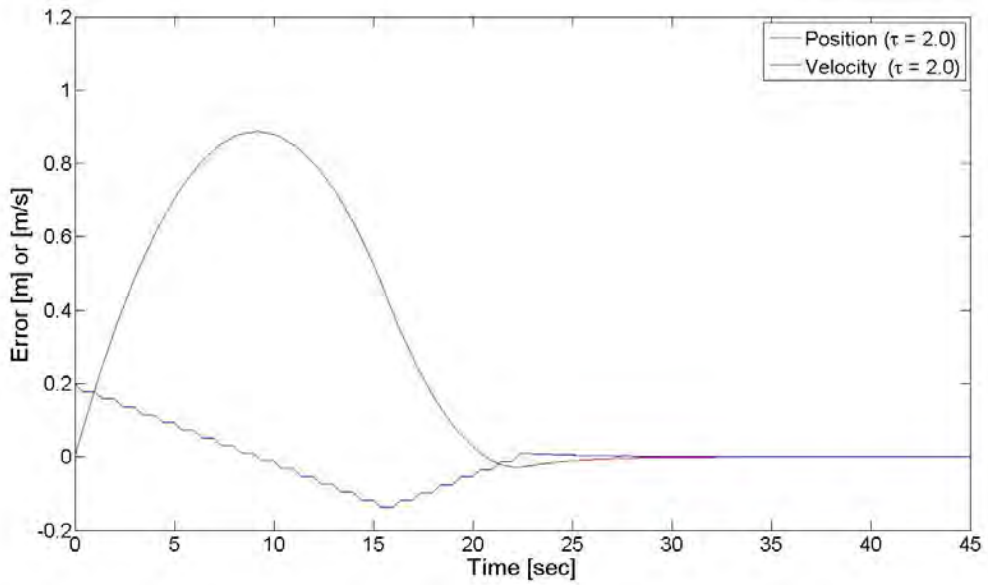

Figure 3.16: Simulation Response of 1-D Translational Errors with Optimized τ

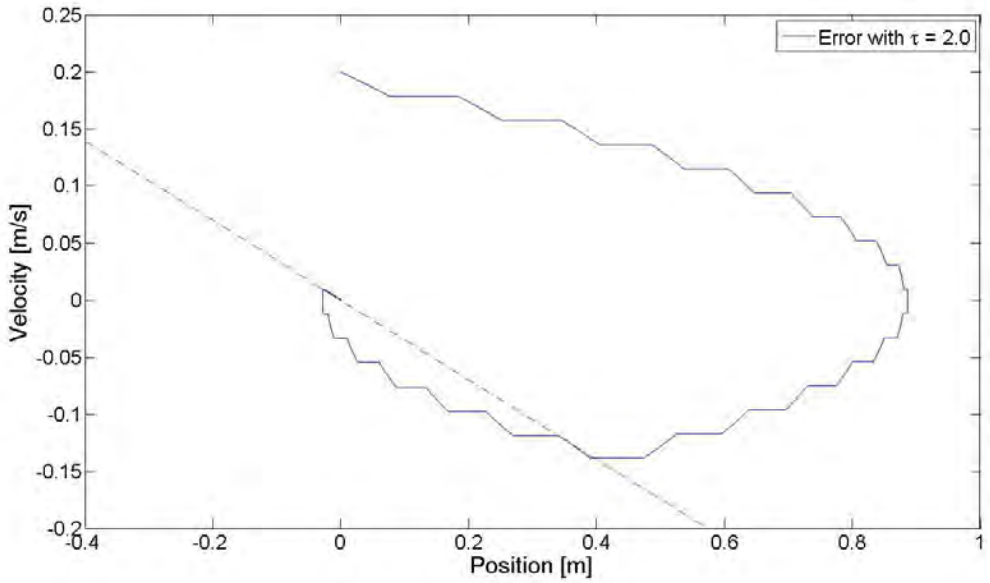

Figure 3.17: Phase Plane of 1-D Translational Errors with Optimized τ

3.2.4 Quaternion Controller. The control algorithm also commands the satellite's orientation through the quaternion controller. This controller is developed through a Lyapunov function to ensure global asymptotic stability[4]. Since the rotational portion of the SPHERES plant is determined by the systems quaternions ($\bar{q}$) and Euler rates ($\bar{\omega}$), it is natural to include these variables in the Lyapunov function (V) to control the plant. The real task however, is inserting these variables in a way to make V positive definite and $\dot{V}$ negative definite. A simple way to ensure the V is positive definite is to make V a sum of quadratic functions.

The output of a quadratic function is always positive for any real input. In regards to the Lyapunov function, quadratic functions are useful because there are no restrictions placed on the function's inputs so long as the variables for the inputs are real numbers. When controlling quaternions, the quaternions and the angular rates are always real numbers, so in fact no restrictions have been placed at this point. With this in mind consider Equation 3.8.

$$V = \bar{\omega}^T \mathbf{MOI} \bar{\omega} + \tilde{q}^T \tilde{q} + (q_4 - 1)^2 \tag{3.8}$$

Equation 3.8 is the Lyapunov function used to develop the quaternion controller. Note that while all terms are quadratic functions, only the last term is easily recognizable as one. The second term is actually the dot product of $\tilde{q}$ with itself which results in the sum of the square of each of the first three quaternion terms. The act of taking a vector dot product with itself can be thought of as squaring a vector much like the input is squared for simple quadratic functions. This of course positive definite. For the first term, recall that the mass moment of inertia is positive definite because the eigenvalues of **MOI** must be positive real numbers. In addition, $\bar{\omega}^T \bar{\omega}$ is also positive definite for the same reason that the second term is. Recall from Section 2.4.3.1 that

[4]While Lyapunov functions are described in detail in Section 2.4.3, keep in mind that desirable Lyapunov functions are positive definite while their time derivative is negative definite [40].

the product of two positive definite matrices is still positive definite. Finally, since each term in Equation 3.8 is positive definite the sum, is also positive definite.

Now that the Lyapunov function is positive definite the next step is to ensure the time derivative of the Lyapunov function, $\dot{V}$, is negative definite. This is done by inserting the appropriate rate equations into $\dot{V}$, simplifying terms, and then choosing an adequate control law to ensure $\dot{V}$ is negative definite.

First, one needs to take the time derivative of the chosen Lyapunov function. Although time is not explicitly listed in Equation 3.8, the quaternions and angular rates are functions of time. Thus, the derivative of this Lyapunov function can be seen in Equation 3.9.

$$\dot{V} = \bar{\omega}^T \mathbf{MOI} \dot{\bar{\omega}} + \tilde{q}^T \dot{\tilde{q}} + 2(q_4 - 1)\dot{q}_4 \tag{3.9}$$

Once the rates for the angular rates ($\dot{\bar{\omega}}$) and quaternions ($\dot{\tilde{q}}$ & $\dot{q}_4$) have been included, substituting those variables with their corresponding equations (Equations 2.50, 2.46, and 2.47 respectively) results in Equation 3.10.

$$\dot{V} = \bar{\omega}^T \mathbf{MOI}(\mathbf{MOI}^{-1}(\bar{u} - \omega^x \mathbf{MOI}\bar{\omega})) + 2\tilde{q}^T(\frac{1}{2}q_4\bar{\omega} - \omega^x\tilde{q}) + 2(q_4 - 1)(-\frac{1}{2}\bar{\omega}^T\tilde{q}) \tag{3.10}$$

The next step is to simplify Equation 3.10, so that the $\dot{V}$ becomes a little more understandable. This is done by first canceling the identity formed by the product of the mass moment of inertia and its inverse in the first term. Next, the remaining terms are expanded out to show how additional terms are canceled out. This can be seen in Equation 3.11.

$$\dot{V} = \bar{\omega}^T(u - \omega^x \mathbf{MOI}\bar{\omega}) + q_4\tilde{q}^T\bar{\omega} - 2\tilde{q}^T\omega^x\tilde{q} + \bar{\omega}^T\tilde{q} - q_4\bar{\omega}^T\tilde{q} \tag{3.11}$$

The terms $\tilde{q}^T\bar{\omega}$ and $\bar{\omega}^T\tilde{q}$ are both the dot product of $\bar{\omega}$ and $\tilde{q}$, and equal the same scalar value. Thus, the second and the fifth terms result in equal but opposite values and hence cancel each other out. Lastly, the product of the third term is zero. Only the first and fourth term of Equation 3.11 remain. The sum of these terms yields the final result as seen in Equation 3.12.

$$\dot{V} = \bar{\omega}^T[\bar{u} - \omega^x \mathbf{MOI}\bar{\omega} + \tilde{q}] \tag{3.12}$$

In order for the system to be globally asymptotically stable, Equation 3.12 must be negative definite over the entire domain so that $\dot{V}$ is valid. While the equation is not obviously negative definite at the moment, it can be manipulated as desired because the control law $\bar{u}$ can still be selected. Therefore any control law that makes the equation negative definite should in theory allow for the SPHERES orientation to be globally asymptotically stable around its commanded orientation when the initial angular rates were zero. Since the control only affects the angular rates, the author chose to include the term K_d to act as a gain that can be adjusted to affect the systems rate. This variable is meant to be a positive scalar and is similar to the derivative gain commonly used in linear PD controllers. Furthermore, the author chose to use the control law shown in Equation 3.13 because these terms canceled out the undesirable system characteristics while inserted the gain, K_d, in a manner that affected the angular rates and made the system negative definite.

$$\bar{u} = \omega^x \mathbf{MOI}\bar{\omega} - \tilde{q} - K_d\bar{\omega} \tag{3.13}$$

When the control law (Equation 3.13) is inserted into Equation 3.12 and simplified, the end result can be seen in Equation 3.14.

$$\dot{V} = -\bar{\omega}^T K_d \bar{\omega} \tag{3.14}$$

Once again, the control law (Equation 3.13) will control the system's quaternions by driving the first of the values of the quaternion error to zero so long as the value for K_d is positive. But what values, if any, give more desirable response characteristics for the quaternion controller? Next the values of K_d should be optimized to yield the most desirable performance characteristics for the quaternion controller.

3.2.5 Optimal Weighting for Quaternion Controller. Now that a control law has be created for the quaternion controller, the variable K_d can be tuned to yield various results. For this application the author chose to consider the response characteristics of the first three values of the quaternion error since these values are suppose to be driven to zero. Specifically, the peak value, settling time, and control usage were analyzed. This section details how the trade study was performed to select the value for K_d that balances these three parameters against each other and then highlights what the author considers to be the best value.

This study began by considering the affects of rotating about one axis at a time. In other words, SPHERES started in line with the inertial frame and was then commanded to roll, about one of its body frame axes. This was done so that rolling about each axis could be looked at individually before analyzing more complicated maneuvers. Commanding SPHERES to roll about its X-axis primarily affects the 3^{rd} quaternion, while commanding SPHERES to pitch up or down primarily affects the 2^{nd} quaternion, and lastly, commanding SPHERES to yaw or rotate about its Z-axis primarily affects the 1^{st} quaternion. In addition all require some input from the 4^{th} quaternion as this quaternion must change to satisfy the constraint that the root sum square of the quaternion vector remains at a constant value of one.

The first test looked at the response of rolling SPHERES 10° about its x-axis. Figure 3.18 shows how the error of the 3^{rd} quaternion changes to meet the requirement. Next, the controller was allowed to run continuously for this example to verify that the control law developed from the Lyapunov equation in Section 3.2.4 was in fact globally asymptotically stable for positive values of K_d.

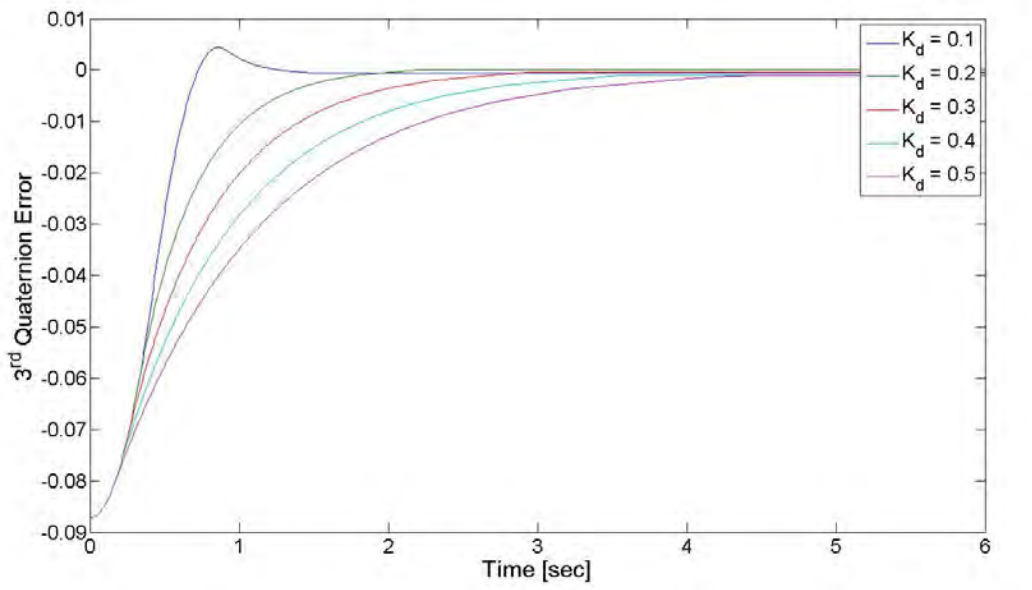

Figure 3.18: SPHERES Response of 3^{rd} Quaternion Error when Commanded to Roll 10° with Full Control

As seen in Figure 3.18, positive values for K_d do in fact provide stable solutions when the controller is permitted to operate the entire time. Furthermore, one can observe a trend in the affect of tuning K_d. When higher values are selected for K_d the response in the quaternion error is more damped. Thus one can think of tuning K_d as changing the damping ratio. But this behavior occurs when the controller is not required to operate in limited time intervals. Figure 3.19 displays what happens when the controller is limited to operate in 0.4 second intervals per second as required to allow for the estimator to run without interference (described in Section 3.2.6.4).

Once the control is limited on time, positive values of K_d no longer guarantee asymptotically stable solutions. Notice how when $K_d = 0.1$, the system is marginally stable. This is because the controller does not have enough damping to operate only 40% of the time and still control the satellite. Therefore the control law is not truly globally asymptotically stable, however for ranges of 0.2 and higher it is still stable. In addition, a larger region of values for K_d display under-damped characteristics with the control limited to 40%. Furthermore when $K_d \geq 0.5$ the response is over-damped

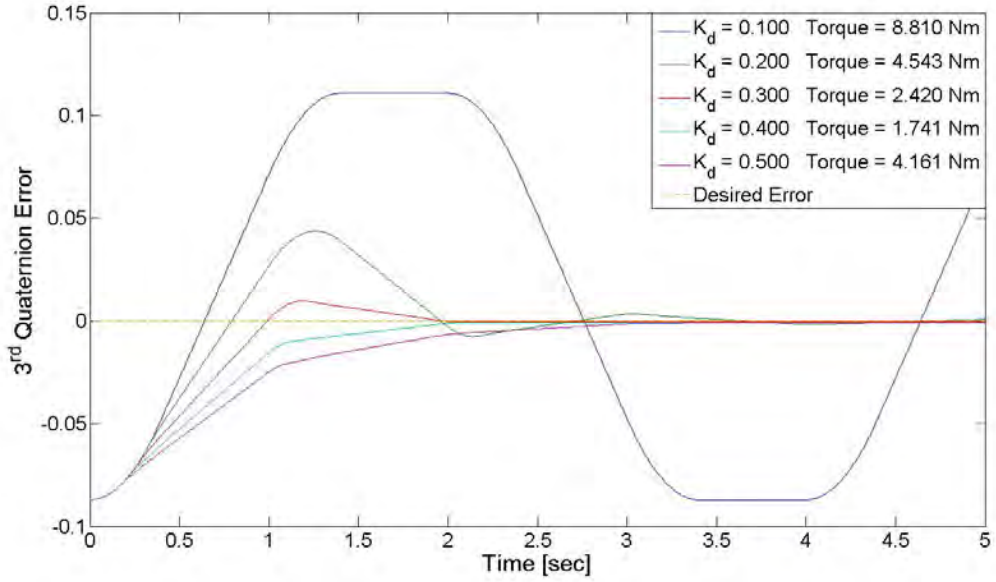

Figure 3.19: SPHERES Response of 3^{rd} Quaternion Error when Commanded to Roll 10° with Limited Control

and the rise time is slower than the rise time of the other over-damped responses that occur when $K_d \leq 0.5$. This limits the range of desirable gains to be between 0.2 and 0.5.

Although the desired operating range of K_d has been limited the responses of the quaternion error vary from under-damped responses to over-damped responses. This makes it somewhat difficult to compare values of K_d as one typically characterizes the response characteristics for under-damped systems differently than one would characterize the response of an over-damped or first-order system. Engineers typically characterize under-damped systems by the percent overshoot and settling time of the response while over-damped responses do not overshoot the desired value and are typically characterized by the rise time (10% - 90%) of the response [37]. Oftentimes the specific application will favor either under-damped or over-damped responses as each have their own advantages and disadvantages. In this application, the two largest objectives for the controller are to complete the maneuver as fast as possible while using the least amount of fuel. As Figure 3.19 shows that values of K_d between 0.3

and 0.4 produce the fastest response while values outside this range take more time to reach a steady-state response. In addition this region also requires less control to perfrom the maneuvers. This can be seen indirectly through the sum of all the external torques applied on SPHERES by the thrusters.

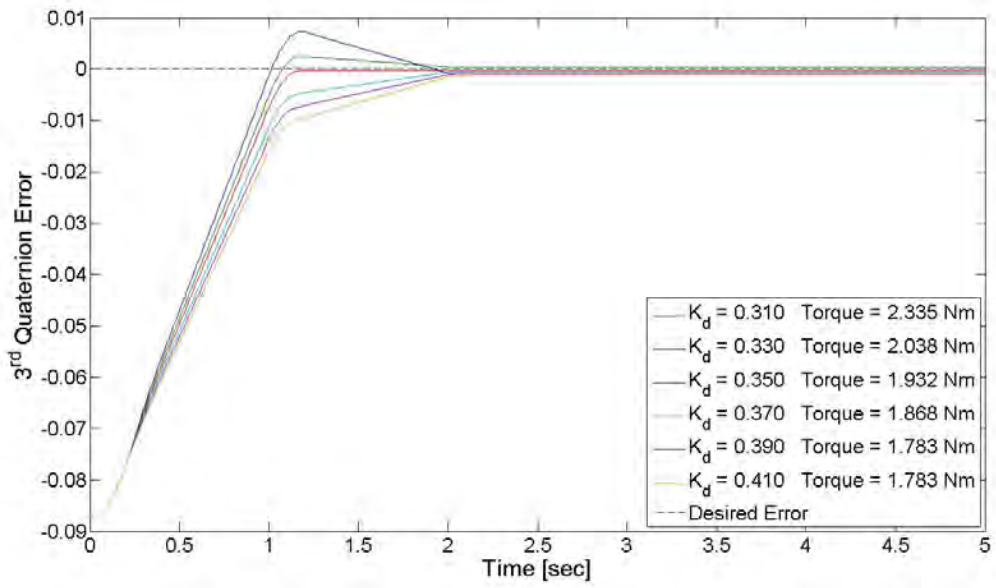

Figure 3.20: SPHERES Response when Commanded to Roll 10°

With this knowledge it becomes apparent that regardless of whether it is an over-damped or an under-damped system, the value for K_d should lie between 0.3 and 0.4. Thus the quaternion controller was run with various values of K_d to observe the response of the quaternion error. Figure 3.20 describes how the quaternion error is driven to zero when various values of K_d within this desired range. At first glance one can see that within this range the value of K_d increases, the amount of torque required decreases, and the response transitions from an under-damped system to an over-damped system. Upon closer inspection the response of the 3^{rd} quaternion error when $K_d = 0.35$ appears to be critically-damped or almost so. This occurs when the damping ratio, ζ of the system response equals one and serves as the transition between under-damped systems and over-damped systems. Controllers are not typically designed to produce critically-damped responses as it is difficult to make a system

that has $\zeta = 1$ exactly. In the case of this controller however, while the over-damped responses required a little less control, the almost critically-damped system reaches its steady state value much faster than either under-damped or over-damped responses. Therefore, the author chose to pick K_d to 0.35 since this response most closely meets the controller's requirements. But this response is only for when the satellite is commanded to roll about its x-axis. What about when the satellite needs pitch or yaw about the other axes?

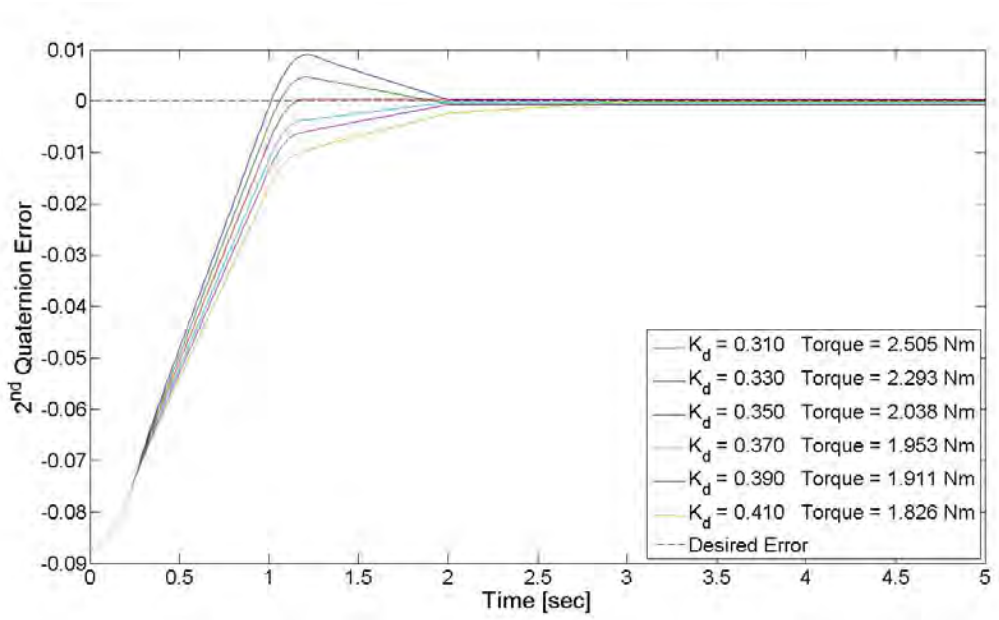

Figure 3.21: SPHERES Response when Commanded to Pitch Up 10°

Figures 3.21 and 3.22 display the response of the appropriate quaternion error when SPHERES is commanded to roll about its other two axes. Both of these figures show that when the desired near critically-damped response is achieved when $K_d \approx 0.35$.

3.2.6 Controller Nonlinearities. In regards to Figure 3.1, the two controllers produce control laws that do not fully account for all the intricacies of the SPHERES system. Specifically, there are a few dynamics inherent to SPHERES that are not accounted for with the current control law. To remedy this, the two control laws are

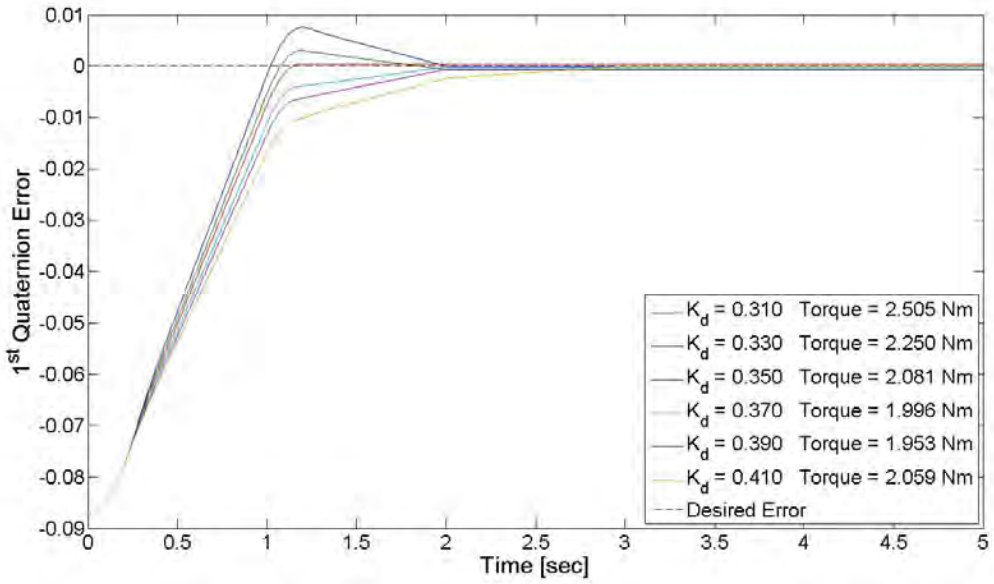

Figure 3.22: SPHERES Response when Commanded to Yaw Right 10°

then modified by the four nonlinearities before the two laws are merged to create the thrust profile described in Section 3.2.7. The four nonlinearities implemented into the control algorithm include a dead-zone, a saturation, a rate limiter, and a periodic signal suppressor to correct processing time allowance.

3.2.6.1 Dead-Zone. The implementation of a dead-zone prevents the system from chattering and wasting fuel. The term chatter describes the phenomena that results from unnecessary control usage. This is particularly noticeable when discrete bang-bang controllers attempt to drive errors to exactly zero. Recall that bang-bang controllers are either completely on or completely off. Thus, while the signal error becomes smaller and smaller, the controller is still only capable of commanding the same magnitude of thrust to drive the small error to zero. This results in an overshoot, that the controller will then correct with yet again, a relatively large thrust in the opposite direction. Figure 3.23 demonstrates this phenomena pictorially.

Figure 3.23 shows a one dimensional case for the relationship between the position and velocity error of a sample spacecraft. Recall that the controller will switch

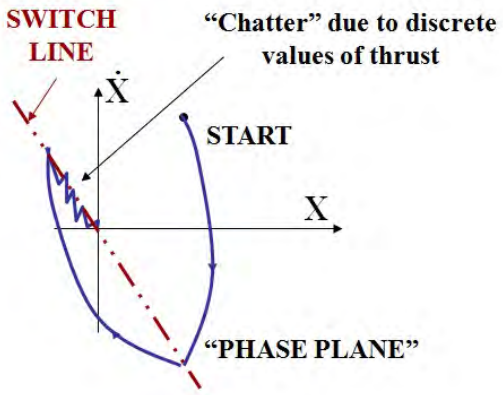

Figure 3.23: Example of Chattering with a Bang-Bang Controller [3]

correction priority, based on the slope of the switch line, between the errors. Eventually, further improvement in the errors results in a need to correct both errors almost at the same time. This can be seen in the second quadrant of Figure 3.23 as the errors almost 'ride' the switch line to the origin. When this occurs the process appears to 'chatter' as the errors approach the origin of the phase plane because the thrusters fire at a set value that cannot be changed during flight. Thus each thrust causes the error to overshoot a little which results in another thrust to oppose the thrust that was just created. Although the error can be minimized by reducing the amount of time the thrusters fire, the control, or fuel, is consumed inefficiently when the system 'chatters' because some control is required to counter the control previously implemented. Thus, a dead-zone is implemented to minimize the effects of chatter by restricting small amounts of control usage. This is implemented by preventing the controller from outputting a signal unless that signal exceeds a specific magnitude [4].

Figure 3.24 graphically displays the describing function of the dead-zone. When the control signal is run through a dead-zone, the magnitude of the control signal is reduced to zero when the signal's magnitude is less than the dead-zone limit, or δ. However, if the magnitude of the control signal is greater than δ then the magnitude is not attenuated. This allows a designer to limit chatter because the controller is unable to provide those excessive counter thrusts.

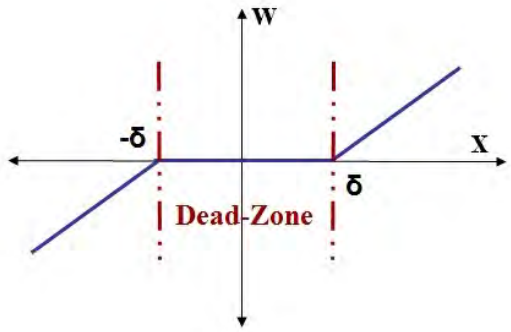

Figure 3.24: Example of Dead-Zone Nonlinearity [4]

Although the prevention of chattering saves fuel use, the implementation of a dead-zone can degrade system performance if the value of the dead-zone it too large. The dead-zone value is the range of the signal that produces no control (2δ). If this range is too high, then the controller is unable to meet the minimum error requirements imposed by the user. Thus, the dead-zone needs to be large enough to prevent unnecessary fuel consumption, but small enough to prevent an unacceptable loss of controller precision. The application of the SPHERES program on the ISS dictate that controller precision is of greater importance than fuel conservation as the SPHERES satellites are typically required to execute multiple maneuvers involving centimeter-level accuracy. With this in mind, the dead-zone has be limited to a range of ± 0.002. This allows the translational controller to command position and velocity values with an accuracy of 0.2 centimeters or centimeters per second respectively. In addition, this dead-zone allows the quaternion controller to drive the quaternion error be within ± 0.002 of the desired quaternion error. Lastly, while this value effectively balances the two competing desires one can change the dead-zone width by changing the variables labeled 'High' and 'Low' in the master code provided to run the simulation. Although the dead-zone is set to ± 0.002 for better accuracy, the analysis performed in Section 4.4 describes how to select the dead-zone to reflect the users needs.

3.2.6.2 Saturation. While the dead-zone conserves fuel by eliminating small control values, the saturation confines the non-zero control values to specific values. This is a practical limitation imposed by the system since the SPHERES thrusters can only maintain a specific thrust value as opposed to a variable value. Therefore, the two control laws become three by one vectors containing either zeros and positive or negative ones. A positive value would correlate to a positive translation or rotation while a negative value would represent a negative translation or rotation. Since the signal will eventually be converted into a thrust value before it is passed into the plant, there is no advantage to be gained by changing the values that the control signals are set to. Therefore, the simulation sets non-zero control values to ± 1.

3.2.6.3 Rate Limiter. Next, a rate limiter is applied to ensure the satellite does not maneuver too quickly. Theoretically, a satellite could continue to speed up by constantly accelerating until the fuel was spent. This means the satellite could reach translational and rotational speeds that would make the satellite dangerous to operate inside the ISS. To safeguard against this a rate limiter is imposed to ensure the satellite stays within acceptable translational and rotational rates. This is implemented using a number of logical ('if/else') commands shown in Figures B.13 and B.14 of Appendix B.2.2. This is accomplished by checking the current rates of the satellite. If the satellite rates are within limits then the control signal passes through as normal. If the rate of the satellite exceeds the specified rate limit then only control signals that correct the problem are allowed while control signals that would exasperate the current rates are set to zero. The translational and rotation rate limits are set by the user. Since no requirements are explicitly stated for suggested or mandatory rate limits, the translational rate limit is set to 0.1 meters per second and the rotational rate limit is set to $6\frac{^\circ}{sec}$.

3.2.6.4 Periodic Signal Suppressor. Lastly, a periodic signal suppressor is installed to cut off the control signal after a specific amount of time. This control

limitation is in place because the accuracy of the SPHERES estimator significantly degrades if the thrusters are firing. This is because the on board metrology system uses ultrasonic noise to determine the satellite's location, and the noise and vibrations produced by the thrusters are strong enough to interfere with this process. In order to prevent this, the controller is cut off a certain percentage of every second to give the estimator uninterrupted time to work. Previous work on SPHERES has shown that the estimator produces adequate results when allowed to run for 0.6 seconds at a time. Therefore, this algorithm follows the same rule of thumb and allows the controller to run for no more than 0.4 seconds per second. The periodic signal suppressor is created in the simulation software by multiplying the control signal with a periodic pulse. The periodic pulse has an amplitude of one and a width of 0.4 seconds.

3.2.7 Controller Signal Logic. Once the controller signals have been passed through the nonlinearities they must be merged together and formatted to create a thrust profile to trigger the thrusters to fire in a manner to produce the force and torque. In a broad sense this is achieved in two parts. First, the control torque signal is used to determine a thrust profile that rotates SPHERES as desired, while the control force signal is simultaneously used to determine a thrust profile that translates SPHERES as the control law dictates. The sum of these two thrust profiles is found and run through a saturation to ensure that the merged thrust profile contains nothing but zeros and ones for the plant to interpret as in Section 2.5.3. The process for determining how the control signals rotate and move SPHERES is similar to process the plant uses to interpret the signal but simply reversed. A number of 'if/else' statements are used to determine if the signals contain positive, negative, or zero value on\about each body axis. This determines whether a positive, negative, or no translation\rotation is required to meet the control laws requirements. Finally, the use of Table 2.1 allows one to build an appropriate thrust vector for each control law as this table relates how rotations and translations and the SPHERES thrusters interact.

3.3 Interface & Simulation

To finish up discussion on the development of the controller, it is beneficial to consider how to interact with this controller to command SPHERES to perform as desired. As expected, any problem with relative motion requires at least two bodies or points to properly describe how one object moves relative to another. Thus, as users specify commands or rules for each of the bodies to obey, they may wish to select desired values for the bodies in different coordinate frames. This allows for simplicity in the design of path planning. Furthermore, path planners might not want to specify every point for the SPHERES to be, rather one might simply specify SPHERES to be at various locations at various points in time. The inputs then must be conditioned and reformatted so that the controller knows what the user is asking for and the user does not have to waste time over defining a path for SPHERES to track. Conditioning the users inputs is split into three sections. First, Section 3.3.1 covers what users can input to command SPHERES. Section 3.3.3 discusses how to interpret user data that is in the global frame. This is referred to as the internal conditioning as this is performed within the simulation for each SPHERES used. Section 3.3.2 contains information needed to convert user commands into arrays the simulation can use that are in the global frame. This is considered the external conditioning since this is performed outside of the SIMULINK® simulation. These calculations are performed before the simulation as they are usually different for each body considered. Lastly, Section 3.3.4 discusses how to use the information attained through each of the simulations to analyze the relative motion between the satellites. For the simulations described in this thesis, the bodies considered were individual SPHERES satellites, and only two were considered to demonstrate the effectiveness of the speed and path control algorithm. However, that does not mean one is unable to adapt the simulation to consider bodies other than SPHERES or more than two objects. This can be modified by conditioning the external inputs as discussed in Section 3.3.2.

3.3.1 User Commands. The controller requires parameterized equations for the desired velocity, and desired quaternions to run. While some path planners may wish to program paths using parametrized velocities and quaternions, various paths are difficult to parameterize and quaternions are typically challenging to visualize. Nonetheless, this method offers the most freedom for users to design desired paths because they directly interface with the controller. This in turn, typically offers the user the best understanding and command of a system. In any case, the author implemented an interface that asks the user for more intuitive information and interprets the results. The master script which runs the simulation to demonstrate the designed control algorithm prompts the user for points of interest. These points of interest are simply points in time that the user wants SPHERES to be doing something specifically. The next sections will discuss how to interpret these points of interest, but for the time being, think of each point of interest as points where SPHERES is being asked to do something new such as speeding up or changing direction. This method allows the user to specify where and what SPHERES should be doing with a relatively small amount of data while the master script interpolates the data to generate the full trajectory. This method for interpolating commands has not been optimized by any means, so other trajectories could exist that allow SPHERES to accomplish the same tasks in less amount of time or fuel. However, since the focus of this thesis is to develop the speed and path control algorithm as opposed to optimal trajectories for the SPHERES program, the author chose to include a simple user interface for path building. Thus, this interface allows one to see how the control algorithm performs. If one wanted to develop optimal paths for SPHERES with speed and path control a new user interface based on user objectives and not specific points in time would likely need to be implemented. In the meantime however, the user needs to record the desired points in time. Table 3.3 contains a brief overview of just how this is achieved and what inputs are required for each run of the simulation. interpolating commands has not been optimized by any means, so other trajectories could exist that allow SPHERES to accomplish the same tasks in less amount of time or fuel.

However, since the focus of this thesis is to develop the speed and path control algorithm as opposed to optimal trajectories for the SPHERES program, the author chose to include a simple user interface for path building. Thus, this interface allows one to see how the control algorithm performs. If one wanted to develop optimal paths for SPHERES with speed and path control a new user interface based on user objectives and not specific points in time would likely need to be implemented. In the meantime however, the user needs to record the desired points in time. Table 3.3 contains a brief overview of just how this is achieved and what inputs are required for each run of the simulation.

Table 3.3: User Inputs for SPHERES Simulation

Input Name	Definition & Purpose	Units	Dimension
Step Size	How often the simulation updates	seconds	(1x1)
Duration	How long the simulation runs	seconds	(1x1)
Time	Array of times of interest	seconds	(nx1)
Position	Array of positions of interest	meters	(nx3)
Pointing Vector	Body frame vector that faces target	meters	(3x1)
Initial Euler Angle	Initial condition for simulation	degrees	(3x1)
Initial Angular Rate	Initial condition for simulation	$\frac{degrees}{second}$	(3x1)
Initial Velocity	Initial condition for simulation	$\frac{meters}{second}$	(3x1)

The Step Size and Duration inputs serve to specify the rate at which the simulation updates as well as the length of the simulation. The underlying solver for this simulation is 'ode3.m'. This fixed-step solver uses the Bogacki-Shampine method to sample the data at each and every time step. As Step Size decreases in value the simulation outputs become more accurate as the simulation begins to mimic the actual continuous system. A Step Size value of 0.0005 was chosen for the simulations run in Chapter IV. The Time and Position variables are all length 'n'. This variable corresponds to the number of points of interest the user decides to include for the simulation. For example, if the user had five points of interest for a particular SPHERES, the Time input would contain the five times at which each of those points of interest should occur. In addition, the Position input would have five rows containing a row vector of the desired position of SPHERES for each point of interest.

Next, the pointing vector allows the user to specify what point of SPHERES should face the target. This could be used to have a camera point to a target or to indicate the side of SPHERES that should dock with a target. The last three variables in Table 3.3 specify the initial conditions for the SPHERES satellite at the start of the simulation. The initial Euler angle is used so the user can provide an initial orientation for the satellite. The Euler angles are used to perform a 3-2-1 rotation sequence commonly called a roll-pitch-yaw sequence. This method was chosen to allow users an intuitive avenue to specify spacecraft orientation. In addition the initial angular rate and velocity values set the constants for the integrators used within the simulation. Although not directly specified, collecting data from the user in this manner also provides the initial position for the first point of interest.

Finally, it is important to note that the last six variables in Table 3.3 need to be input for each SPHERES satellite that will be used for each simulation. Thus when considering the relative motion between an inspector and a target, two SPHERES would need to be included and two sets of variables would be needed since these satellites will likely not have the same requirements for the entire simulation. One can see how this interface is implemented within the simulation by referencing the m-file labeled 'SPHERES_simulation.m'. This file is located in Appendix A.

3.3.2 External Conditioning. One should note that while the satellites of SPHERES program each have Time and Position inputs for each of their respective points of interest, these values are recorded in the global frame. Although all satellite points of interest are recorded in the global frame, the target satellite needs its position from the origin of the global frame and the simulation asks for the inspector satellite points to be the desired range from the target. This is because users typically are not interested where the inspector satellite(s) is with respect to the global frame so much as they are concerned with where the inspector is with respect to the target. When using multiple satellites, Section 3.3.2.1 details how to interpret data for the target satellite (or satellites if one was to change the inspector's objective within a

simulation) are discussed in Section 3.3.2.1 while details about interpolating data for the inspector satellite(s) is discussed in Section 3.3.2.2. Regardless of what category the SPHERES satellite falls into, the user input data will need to be interpolated to create a full set of data points for the simulation because the user's points of interest are not expected to exist at every time step in the simulation. Within the MATLAB® master script, this routine is performed in the sub-function labeled 'datainterp.m' and can be found in Appendix A.2. In a broad sense this file receives the users desired values for each point of interest and develops time and velocity arrays that contain values for every time step in the simulation. The 'datainterp' routine accomplishes this task by receiving the Step Size, Duration, Time, Position, and initial Euler angle variables mentioned in Table 3.3. The simulation time array is created first with the Step Size and Duration variables. Next, the time of each of the points of interest are matched with the same time values in the simulation time array. This process makes it possible to determine where the position and Euler angles for the points of interest fall into the simulation Time array. With this knowledge the position vector can be determined for all values of time in the simulation time array. This is done by linearly interpolating for the missing points of these variables. A linear interpolation is used to fill in the data between the points of interest because user is not explicitly interested in what happens between the points of interest. If the user truly was interested, then they would have specified more points. It is possible that other forms of interpolation could be used to optimize the path between the users points of interest but that study is beyond the focus of this thesis. Returning to interpolating the data, recall that a line can be defined in slope intercept form as seen in Equation 3.15 where the variable,'$\bar{x}$', is specified by '$\bar{m}$', the slope, the time 't', and the intercept '$\bar{b}$'.

$$\bar{x} = \bar{m}t + \bar{b} \tag{3.15}$$

Each section between the points of interest has a distinct value for $\bar{m}$ and $\bar{b}$. To calculate the slope, the time and vector component of the current point of interest

(denoted by the subscript 'i') and the next point of interest are used as shown in Equation 3.16.

$$m = \frac{x_{i+1} - x_i}{t_{i+1} - ti} \tag{3.16}$$

Once the slope has been calculated for each component in the position vector, the intercept term is calculated using the current time, vector component, and slope to solve Equation 3.15. Once the slope and intercept terms have been found for each of the gaps between the points of interest, Equation 3.15 is used to solve for each of the vector components for the position array. Next,the control algorithm needs the velocity array. Fortunately, since the position vector is already parameterized the velocity array can be found for each time step by taking the derivative of the position with respect to time. Recalling Equation 3.15, this derivative is simply the slope. Thus the velocity array is simply $\bar{m}$. This holds true as long as the initial slope intercepts are recorded as the initial conditions for the integrators within the simulation as discussed in Section 3.3.3.

Subsequently, the 'datainterp.m' routine converts the initial Euler angle into an initial set of quaternions. This is achieved by using the initial Euler angles to perform a 3-2-1 rotation using Equation 2.14. The rotation matrix is then used to determine the first eigenaxis of rotation along with the principal Euler angle using Equations 2.21, 2.22, and 2.23. At this point the initial quaternions are derived from the eigenaxis of rotation and the principle Euler angle using Equation 2.26 and Equation 2.27.

Although the 'datainterp.m' subroutine prepares the user specified information for the simulation, more signal condition needs to be performed before the simulation can be run. This is because the target and inspector spacecraft have slightly different inputs that need to be accounted for.

3.3.2.1 Target. The velocity information for the target is input with respect to the global frame. Thus, the velocity array can be directly imported into the SIMULINK® simulation. Furthermore, the plant has four initial condition vectors: position, velocity, quaternion, and angular rates. The initial conditions for the angular rates simply need to be converted from degrees per second to radians per second, and the initial conditions for the position and velocity are specified by the user. Lastly the initial conditions for the quaternions are supplied from the 'datainterp.m' subroutine.

3.3.2.2 Inspector. Formatting all the variables for the inspector satellite(s) is more involved as the user supplies this information in relative to the target. Thus, the information must first be converted to produce the inspector's position vector in the global frame. Once all the information is in the global frame, formatting the deputy's user-specified information is identical to that of the target's information discussed in Section 3.3.2.1. Therefore this section only focuses how to create the inspector's desired position vector.

Since the user specifies the desired range the inspector should be from the target, the sum of the target's position and the range should result in the desired position vector of the inspector. This is a valid approach as long as the correct position vector of the target is specified. Figure 3.25 illustrates this concept in two dimensions.

Figure 3.25 displays the relationship between two SPHERES satellites at an arbitrary point in time during a simulation involving a target (T) and an inspector (In) satellite. In addition, it should be noted that all vectors are represented in the global frame. The user supplies the desired location of the target, $\tilde{r}_t$, and the desired range the inspector should be from the target SPHERES, $\tilde{\rho}$. Additionally, the target's true location, $\bar{r}_t$ is also shown. Before the control algorithm is applied to the inspector, the user inputs need to be adjusted to reflect the true desires of the user. This is because there exists a potential that the target is not actually where it is supposed to be at any point in time. This error is denoted as $\bar{e}_t$ in Figure 3.25. Regardless of how small this error is, the error will affect the user's input because $\tilde{\rho}$ is dependent upon where

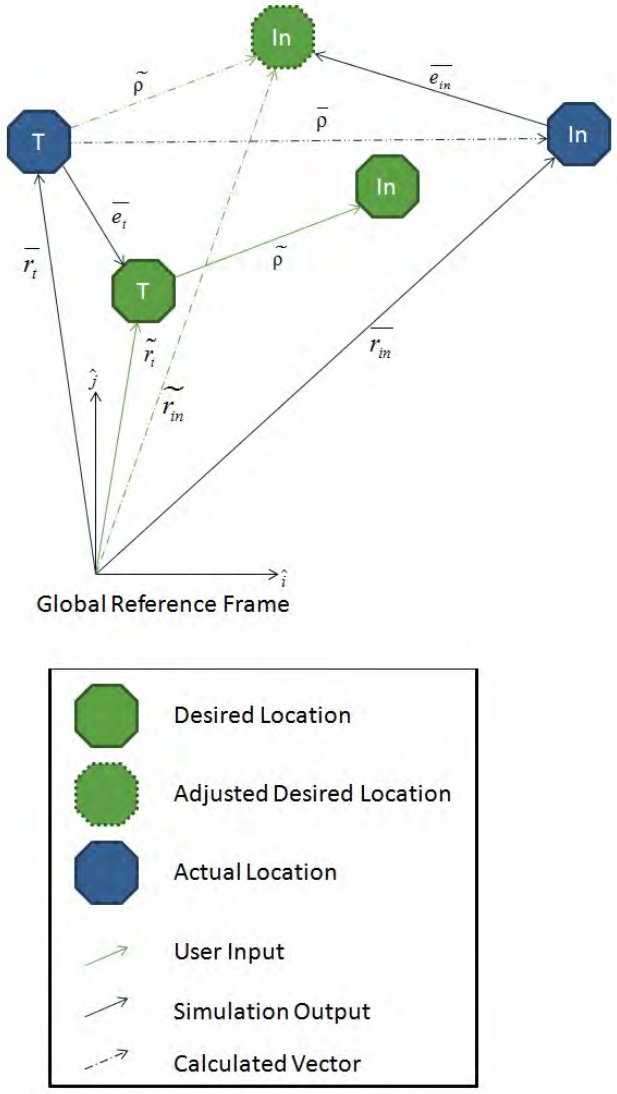

Figure 3.25: 2-D Illustration of Correction of Desired Position

the target actually is instead of where the target satellite should be. Thus, the desired position vector of the inspector, $\tilde{r}_{in}$, is the summation of the actual location of the target and the desired range the inspector should be from the target. At this point, it is worth considering that this simulation requires the target to be simulated before the inspector is simulated to get $\tilde{r}_{in}$. This is because the simulation does not consider the SPHERES estimator during simulation. In reality, the SPHERES estimator is capable of determining the position and velocity of the target as well as the inspector. Thus the inspector does not require full knowledge of the target for every point in time like the simulated SPHERES needs because the estimator provides the required information real time. In either case, one should verify that the true position of the target is used instead of the desired location to ensure that inspector SPHERES follows the correct path. Once this is done, the initial conditions for the inspector must be called. The initial conditions for the position, velocity, quaternions and the angular rates are found in the same manner as those modified for the target.

3.3.3 Internal Conditioning. As the simulation is running the user specified values are called for each time step. This is performed in the user commands block. The user commands block in Figure 3.1 reads the desired target position and velocity from look-up tables that were filled with the quaternion array and velocity arrays developed in Sections 3.3.2.1 & 3.3.2.2. Next, the velocity information is integrated to provide a desired position as well as a desired velocity. The integrator also contains the value of the initial intercept for the position vector to ensure the desired values line up as the user initially specified. Although users could have inserted both desired positions and velocities into the simulation without using the integrator, the author chose just to import the desired velocity for a few reasons. First, the values for position and velocity are defined with respect to time, thus only either position or velocity is really required, as the other can be found through differentiation or integration respectively. Secondly, the velocity was selected to be imported over the position because numerically integrating is less prone to errors than numerically dif-

ferentiating. Lastly, while the linear interpolation performed in Section 3.3.2 does use an easy derivative, future path planning interfaces may not ask for inputs in position at all. Thus this control algorithm was designed with the future path planning implementations in mind, and only asks for the desired velocity information so that easily incorporated with other path planning techniques relatively quickly. In any case, the user commands block in Figure 3.1 outputs the desired velocity and position of the target in the global frame. The determine errors block further conditions the user's inputs by rotating the finding the difference in the translational states and determining what quaternions are needed to point to the target. Section 3.1 discusses how this was performed, and completes the process for the design of this speed and path control algorithm.

3.3.4 Post-Processing of Relative Information. Once both target and inspector simulations have been run, the results are processed one last time in order to extract the information on the relative motion of the satellites. Recall that the control algorithm minimizes the error in the relative motion of the satellites, yet the SIMULINK® component of the simulation generates the position and velocity vectors of SPHERES in the global frame. To find the relative motion of the satellites these vectors must first be manipulated to generate the desired information. In addition, the user information needs to be manipulated correctly before the desired values can be loaded into the simulation. Recall that Figure 3.25 provides a brief two-dimensional depiction of vectors the user supplies and provides a foundation to understand how to interpret the desired results. In addition, this figure only shows the position vectors and does not include the satellites' velocity vectors so that the plot can be easily understood. The control algorithm generates $\bar{r}_t$, or the actual location of the target satellite when the algorithm is applied to the target. Yet in order to compare a user's desires with the actual results of the simulation $\bar{\rho}$, or the actual distance that the inspector needs to be from the target must be calculated. This is achieved by finding the difference between $\bar{r}_{in}$ and $\bar{r}_t$. Once $\bar{\rho}$ has been identified for each point in

the simulation, the actual relative motion of the inspector can be compared with the desired relative motion specified by the user.

This concludes the discussion of the methodology used to create both the control algorithm and the simulation. The control algorithm incorporates two controllers. The translational controller maintains the position and velocity of each satellite in the global frame using a bang-bang controller with optimal weights provided by an LQR. Additionally, the quaternion controller maintains the satellite's orientation by ensuring the derived Lyapunov function is asymptotically stable. The various gains used throughout this control algorithm have also been optimized to minimize the transient response of the system. Specifically, the percent overshoot and the settling time of the system was minimized. This method allows for each error to be reduced as quickly as possible so that error does not grow over time with each new desired input. The simulation was developed to compare the user inputs with the satellite state vector supplied by the plant. This comparison results in the translational errors and the quaternion error. The translational error is rotated from the global frame into the satellite body frame before being inserted into the control algorithm. The quaternion error is found using the eigenaxis and principal Euler angle method of rotation to compare the difference between where the satellite is pointing and where it should be pointing. These errors are applied to the control algorithm with generates two control laws, one for each type of error. The control laws are modified by four nonlinearities to meet constraints fundamental to the SPHERES program. Specifically, a dead-zone is applied to improve fuel consumption, a saturation is included to ensure the thrusters fire at a specific value when they do fire, a rate limiter is enforced to ensure the satellites operate at safe speeds inside the space station, and a periodic signal suppressor is activated to cut the control signals off after 0.4 seconds per second of operation. These control laws are then used to generate a thrust profile to update the satellite state vector using the system dynamics found in the plant. Now that the methodology for this research has been discussed, the control algorithm is simulated in Chapter IV to demonstrate the controller capabilities.

IV. Results

Following the methodology described in the previous chapter, the results have two primary objectives. The first task is to demonstrate that the speed and path controller works through simulation, and the second task is to provide an analysis of how the dead-zone implemented on the control signals can be adjusted to improve either accuracy or fuel efficiency. Section 4.1 discusses how the speed and path controller is to be verified while Section 4.2 & 4.3 validate that the model successfully performs functions required for speed and path control through simulation. Section 4.4 analyzes the relationship between the dead-zone nonlinearity and system performance. Lastly, Section 4.5 provides a summary of the results of this research and provides a brief application for how these results can be applied in future research.

4.1 Model Verification

As each component of the simulation is created, small tests are run to ensure each component performs as desired. This allows the designer to verify that each part of the simulation is correct. In this way, each subsystem is tested to ensure it generates appropriate outputs for given inputs. To elaborate, consider the verification of the subsystem used to determine the R_{bi} from a set of quaternions[1]. This subsystem converts the quaternions from the satellite state vector and outputs the rotation matrix associated with those quaternions, and is based on Equation 2.29. The general procedure to verify the model consists of inputting quaternions with known rotation matrices and checking to ensure the generated rotation matrices matches the expected value. First, the quaternion vector shown in Equation 4.1 is applied.

$$\bar{q} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.1}$$

[1]This subsystem is shown in Figure B.4 of Appendix B.

This quaternion vector results when no rotation occurs, or when the rotation matrix is equal to the identity matrix. After the vector is applied through simulation, the subsystem generates the identity matrix. This indicates that the functions runs and is mostly correct but this test is not able to indicate whether the rotation matrix takes information from the inertial frame and converts the information into the body frame or if the rotation matrix does the reverse since $R_{bi} = R_{bi}^T$. Thus, another test is needed to resolve the uncertainty. The second test involves a set of quaternions that result from rotating the body frame 30° about the third axis. The rotation matrix generated from this test is then multiplied with a unit vector along the first axis of the inertial frame. The rotation matrix should then convert this vector to be represented in the body frame coordinates if indeed this matrix is R_{bi} which is found prior to the test. After running this test the rotation matrix is found to successfully rotate an inertial vector into body frame coordinates proving that this matrix is indeed R_{bi}. Similar tests are performed to each other subsystem to ensure each component performs as expected. Once each subsystem is integrated into the entire SIMULINK® simulation, one must verify that correct coordinate frames are being used throughout the simulation. Figure 4.1 provides a quick reference to understand which coordinate frame is being used when and where.

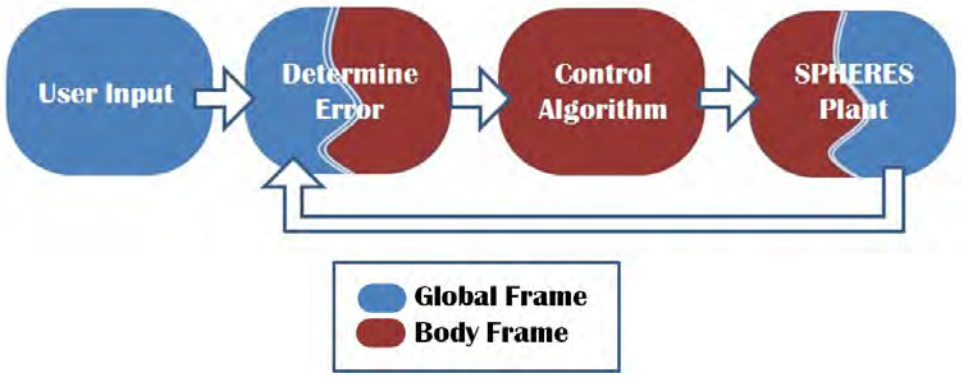

Figure 4.1: Coordinate Frame Flow Diagram

As Figure 4.1 suggests, the user inputs data in the global frame, and the control algorithm is applied in the body frame. The 'Determine Error' block receives global frame information and provides body frame information for the control algorithm. In addition, the 'SPHERES Plant' block accepts a body frame thrust profile and converts the corresponding forces and torques into the global frame before the states are updated. Keeping track of which coordinate frame is used and how it is applied makes it possible for the simulation to be verified as a whole.

Once the subsystems are brought together to create the entire simulation, one should verify the entire system. The simplest method for verification involves applying a series of tests of increasing complexity. The simplest test include running the simulation while trying to keep the satellite stationary. Next, one introduces a rotation, then a translation, and then a translation with a rotation. An exhaustive description of these tests are not included within this thesis because the author believes their inclusion would detract from the bigger picture of determining if the control algorithm is successful. Yet, one should not devalue to usefulness and necessity in testing each subsystem just because this step is excluded. In either case, once the control system is shown to work, the algorithm must be validated to ensure the controller is capable of providing the user with the desired results. This is achieved through the use of an example test involving two SPHERES. One acts as a target and the other acts as an inspector. Section 4.2 explains the purpose and goal of this simulation, and Section 4.3 provides the results of this simulation.

4.2 Simulation Description

For the simulation described in this section, two SPHERES are used. The first SPHERES acts as a target, follows a straight path, and does not change its orientation. The second SPHERES acts as an inspector and is commanded to have a particular body frame vector face the target at all times. This body frame vector, or pointing vector, would likely represent a camera or other sensor that needs to be directed at

the target. The desired path of the inspector is relative to target being viewed. This path is shown in Figure 4.2.

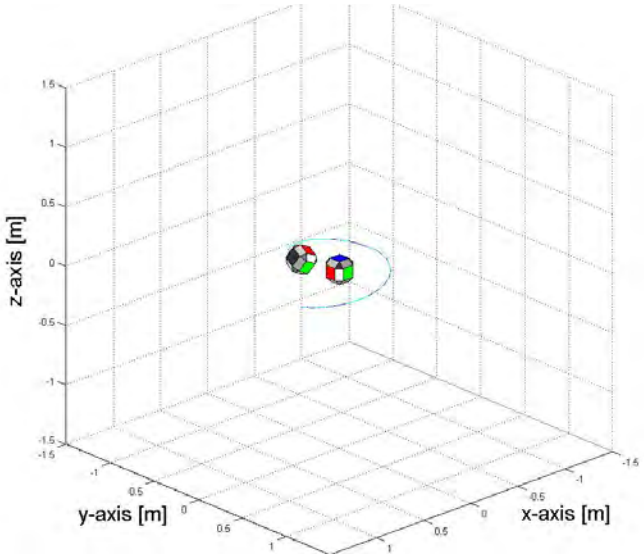

Figure 4.2: Relative Path of Inspector SPHERES

For this simulation, the inspector is tasked to collect information on all sides of the target. This is accomplished by flying a circle in the X-Y plane of the global frame while translating along the Z axis. In addition, the inspector is tasked to maneuver slower along the first 135° of the circle before speeding up to complete the path in the desired time. The purpose of this simulation is to show that the controller is capable of varying speeds along the path while minimizing the position, velocity and pointing errors. The control algorithm is considered successful if the target stays on its path without rotating and if the inspector maintains the desired path and speed along its path while pointing the sensor to the target.

4.3 Simulation Results

The simulation is designed to validate the control algorithm by demonstrating the controller capabilities. Particularly this simulation is run to highlight the control algorithm's ability to command the satellite to maintain a specific path and speed while simultaneously pointing at a target. Furthermore, this simulation demonstrates

the controller's capability to translate SPHERES without changing the satellite orientation because the target is required to translate without rotating. The body frame coordinate system is shown for each SPHERES is depicted on the panels of each satellite. The panel on the positive x-axis is shaded red, the panel on the positive y-axis is shaded green, and the panel on the positive z-axis is shaded blue. In addition to this, a sensor has been placed on positive z-axis of the inspector satellite. The sensor field-of-view is illustrated as a yellow cone emitting from the location of the sensor. Lastly, the simulation plots the satellite paths as the satellites move through them. The desired path of the inspector has a cyan color while the actual path the inspector takes is colored blue. The desired path of the target is shown by the magenta line and the actual path of the target is displayed in red. Figure 4.3 pictorially describes the initial phase of the simulation, showing the inspector slewing to the target.

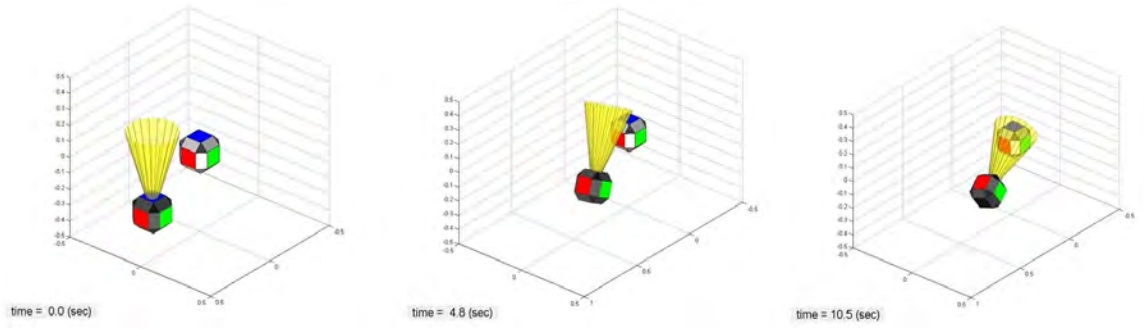

Figure 4.3: Initial Phase of Simulation

Both SPHERES began with their coordinate frames aligned with the global frame. This means the sensor is not facing the target. Thus, as the inspector begins to perform the inspection, the satellite must rotate to face the target. This is can be seen at time progresses to ten seconds. After ten seconds the inspector sensor is pointing towards the target as commanded. The inspector takes ten seconds to point to the target because the angular rate of rotation is limited to six degrees per second, and due to the path requirements and the initial conditions, the sensor begins pointing approximately sixty degrees away from the target. To improve this time, one could either relax the constraint of the angular rate limiter or one set the initial

conditions of the inspector such that the pointing error of the sensor begins with a smaller number. In any case, the initial phase of this test shows that the control algorithm is capable of pointing the satellite to meet requirements. The remaining portion of the test is shown in Figure 4.4.

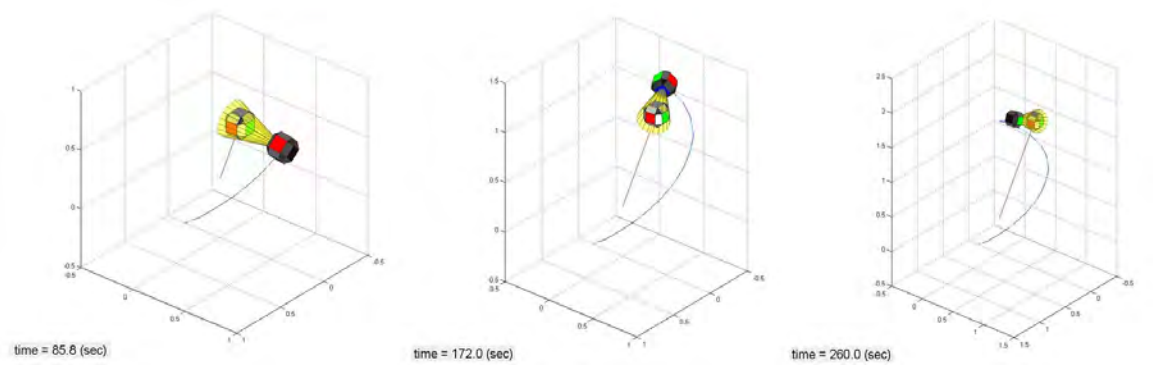

Figure 4.4: Simulation of Satellite Inspection

After the inspector directed the sensor to the target, the satellites maintains the correct orientation throughout the remaining portion of the test. The inspector also travels around the target and maintains the correct speed and position along the path. Recall that the inspector is tasked to fly around the target along the path shown in Figure 4.2. Since the target is moving however, the desired path of the inspector the path appears to stretch out along the target's path. This effect is necessary to ensure the relative path of the inspector is the same are the path requested by the user. Yet if one looks down the path taken by the target then the circular path of the inspector is revealed as in Figure 4.5.

The results from the simulation epitomize the capabilities of the speed and path control algorithm. Further examination of the inspector pointing error indicates that the control algorithm can successfully be used to point a sensor as desired. To illustrate this, Figure 4.7 displays the inspector pointing error (or principal Euler angle) as a function of time.

After the inspector satellite maneuvers the sensor to focus on the target the pointing error never exceeds two degrees. This demonstrates that the control algo-

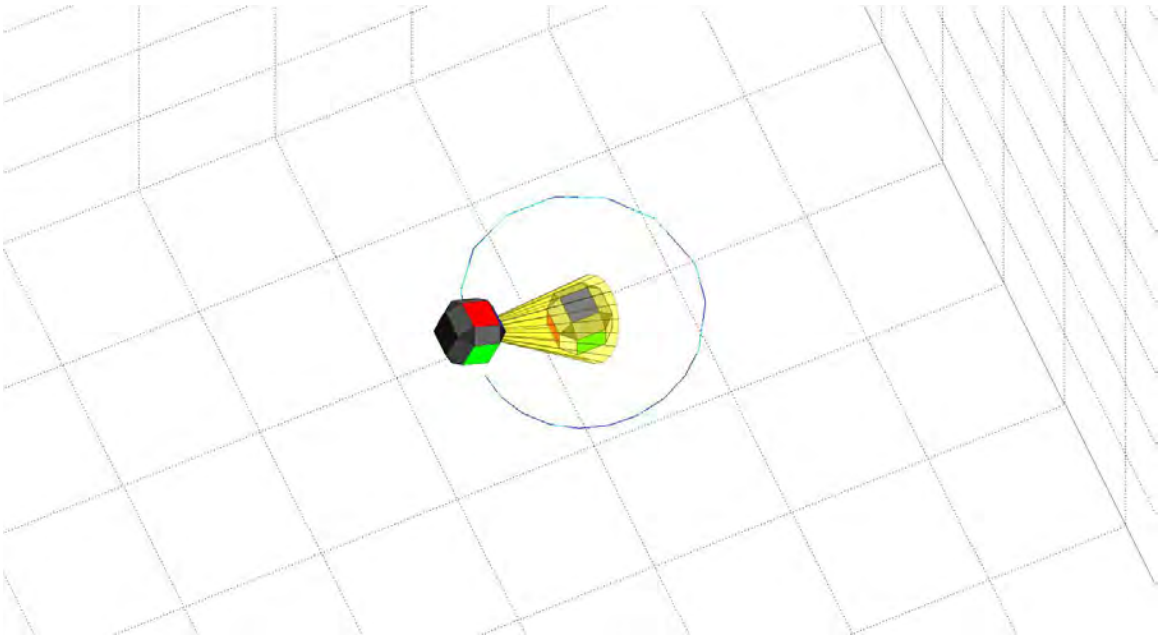

Figure 4.5: Path of Inspector Satellite

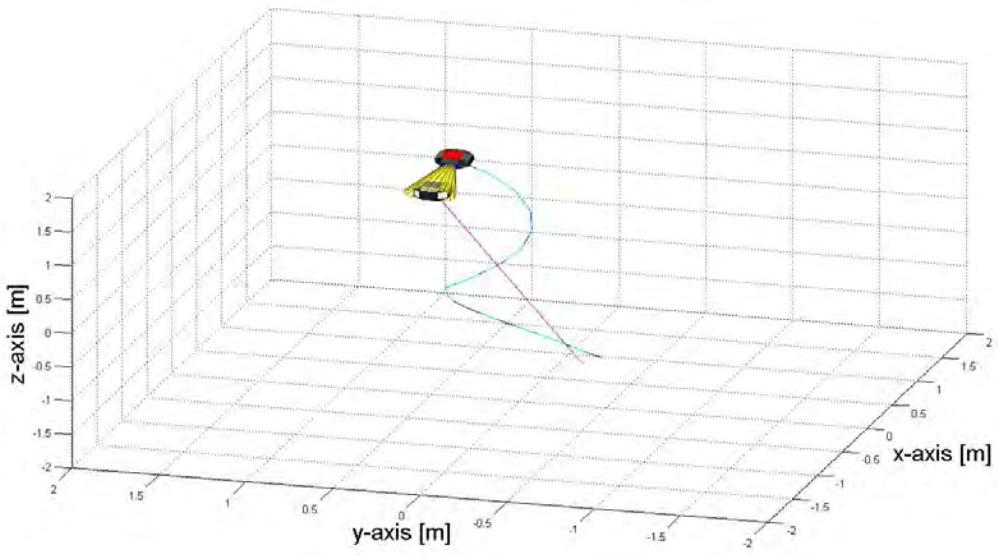

Figure 4.6: Simulation of Satellite Inspection with a Moving Target

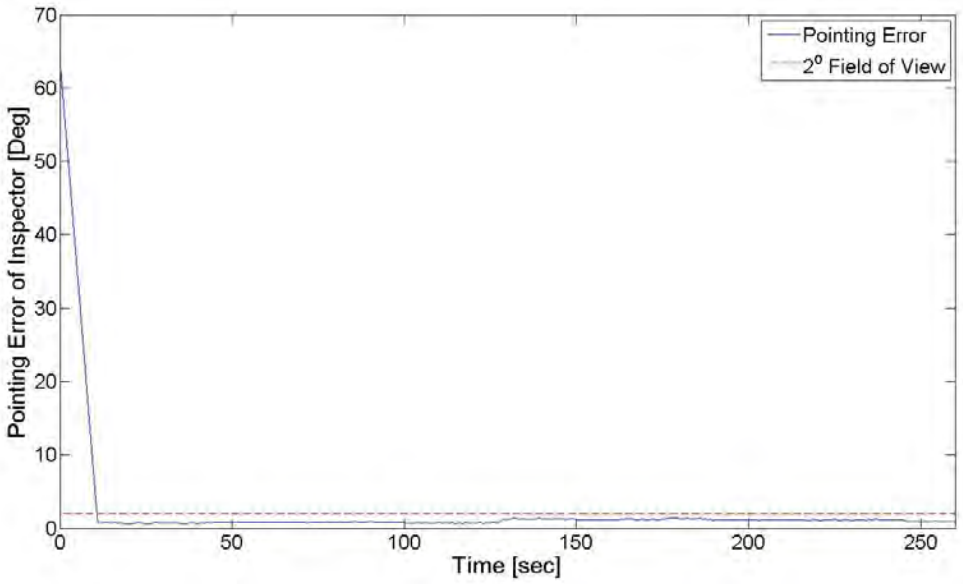

Figure 4.7: Inspector Pointing Error

rithm is capable of reducing the control error described in Figure 3.2. Although jitter can be observed in the pointing error displayed in Figure 4.7, once the satellite has directed the sensor, this jitter only results in an average of one degree of error, which is well within the requirements for most sensor applications that would be installed on a satellite with the same properties as SPHERES. The quaternions and angular rates of the inspector also reveals that the rate limiter prevented the controller from spinning the satellite too fast.

As depicted in Figure 4.8, the angular rates of the inspector never exceed six degrees per second. In addition, once the sensor is pointed to the target (approximately ten seconds) the quaternions behave in a sinusoidal fashion. This is as expected since the inspector is circling around the target to keep the target in the sensor field of view. The angular rates also appear to chatter throughout the simulation. This is because the dead-zone is purposely small to provide more accuracy. The trade-off is seen though through the chatter in the angular rates because the chatter results in more fuel use. Selecting a dead-zone to meet mission needs is further discussed

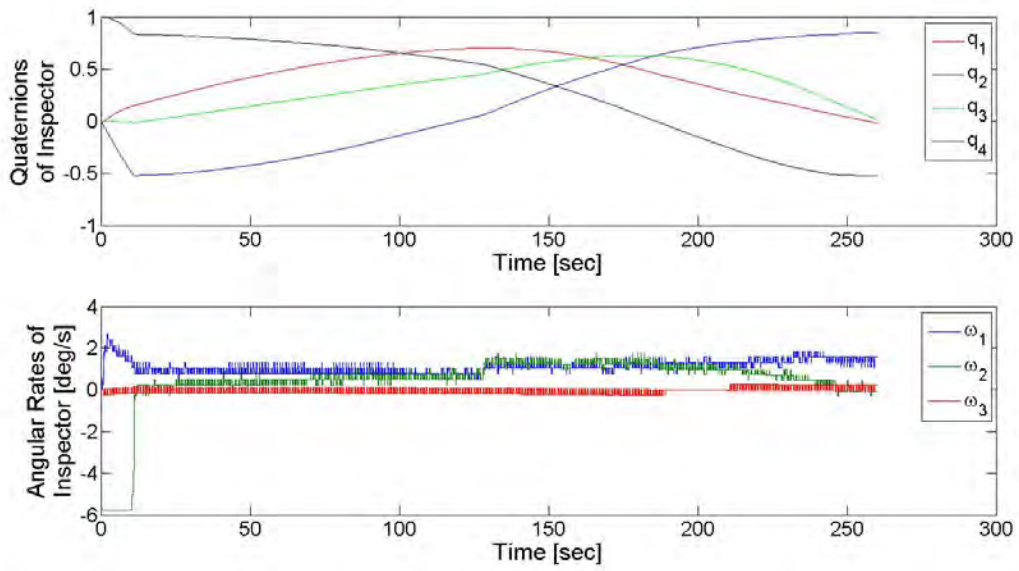

Figure 4.8: Quaternions and Angular Rates of Inspector Satellite

in Section 4.4. Another dynamic of this system is observed when interrogating the angular rates.

Figure 4.9 provides a close up view of the angular rates during a portion of the this test. At this detail, the angular rates are easily observed to remain constant for a time before appearing to chatter again. This is due to the periodic signal suppressor designed to 'kill' the control signal after 0.4 seconds of every second. As a result, the angular acceleration to zero for 0.6 seconds of every second. When this occurs the angular velocity remains constant until the controller is allowed to run for the next 0.4 seconds. Next, the quaternions and angular rates of the target are displayed in Figure 4.10.

Recall the target is commanded not rotate as the satellite translates. This is achieved by commanding the negative z-axis of the target to always point straight down. As one can see from Figure 4.10 not much is happening throughout the simulation. Although the angular rates are adjusted to counteract small deviations, the satellite's orientation never has any noticeable changes. The small deviations are initially attributed solely to the fact the the principal axes of the satellite are not per-

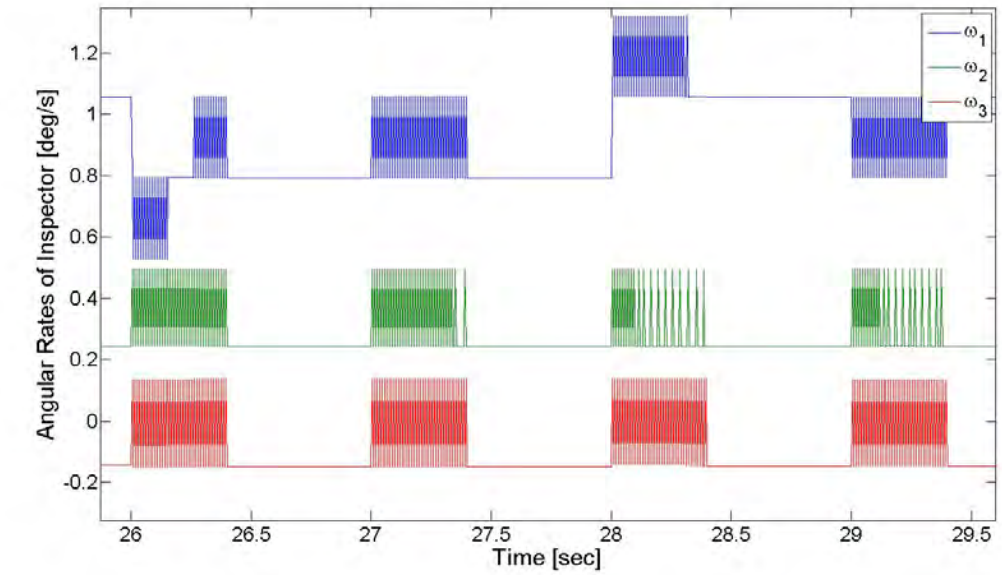

Figure 4.9: Periodic Signal Suppressor Affects on Angular Rates

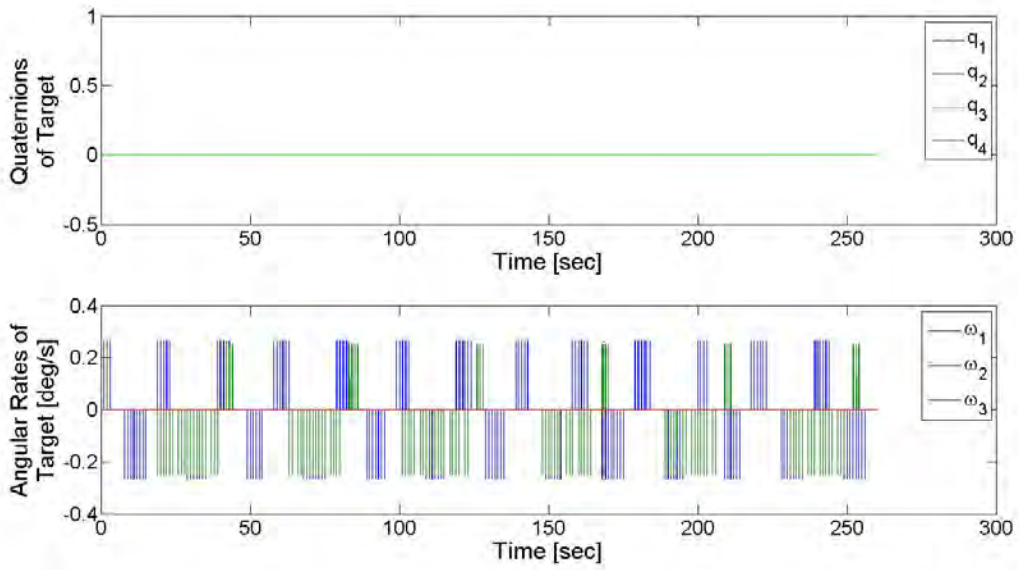

Figure 4.10: Quaternions and Angular Rates of Target Satellite

fectly aligned with this satellite body frame. Then as the angular rates are adjusted through thrusting to compensate for this, opposing thrusts are applied to counter the recently applied angular rates. Nonetheless, Figure 4.10 illustrates that this speed and path control algorithm is capable of translating a satellite without changing the satellite's orientation.

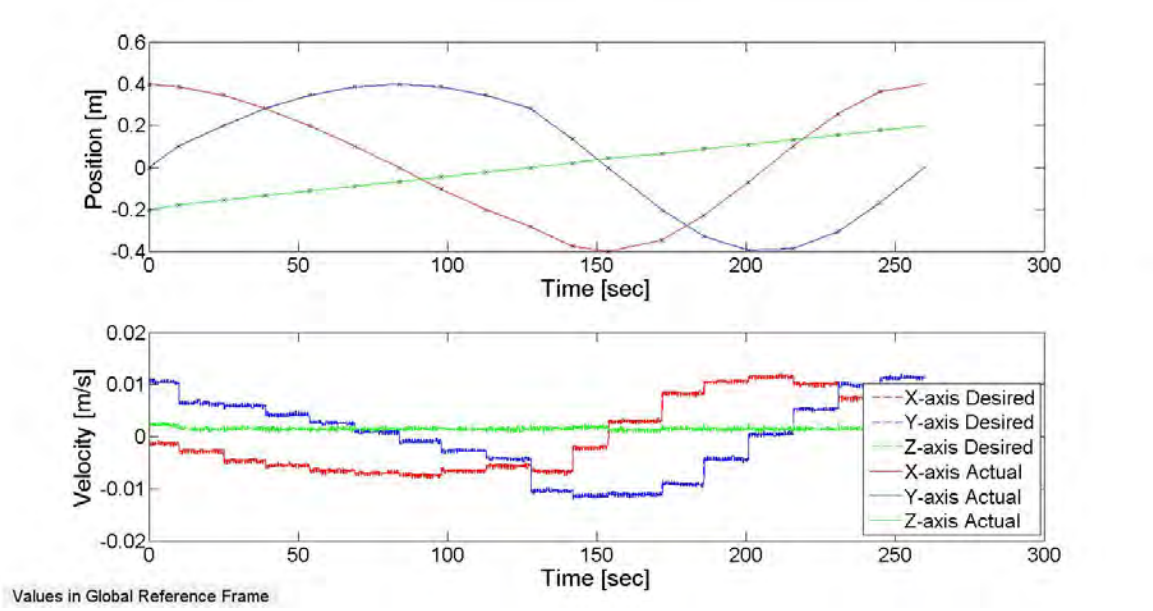

Figure 4.11: Relative Motion of Inspector

Next, Figure 4.11 displays how the inspector translates relative to the target. The 'x's placed along the plot's relative position indicate the user specified position for each specified point in time. Take note that the magnitude of the velocity values increase after 128 seconds. This is because the satellite was tasked to speed up along the x and y axis of the global frame at this point in time. The change in sign of the velocity vectors simply indicate that the satellite is moving around the other side of the target. The effects of the dead-zone and signal suppressor can be observed through the velocity vector as well. Furthermore, the position error never exceeded one millimeter for this test. Since this test had the inspector start in the correct location however, another was run to ensure that the controller could correct the translational errors as oppose to simply maintaining them. In this test, the

initial condition was such that the inspector begins with an error of 0.28 meters. Figure 4.12 shows how the translational errors change with time when this test is performed. This figure demonstrates that the inspector never deviates more than two millimeters from its desired position relative to the target after fourteen seconds have passed. In addition, the velocity never exceeds the imposed translational rate limit of ten centimeters per second. Furthermore, once the initial velocity error is corrected, the velocity error never exceeds one millimeter except for very short periods of time in which case the velocity error does not exceed six millimeters per second. Thus, the results mentioned herein validate the controller designed within this thesis. Specifically, the controller is capable of allowing an inspector to track a target while maneuvering along a prescribed path. In addition, the controller is capable holding a specific orientation while translating.

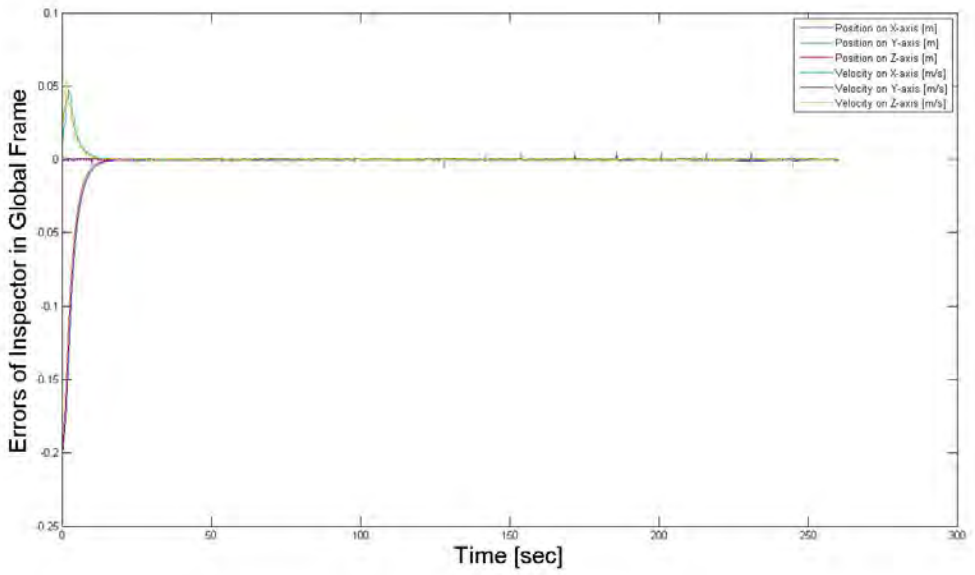

Figure 4.12: Relative Motion of Inspector

4.4 Relationship Between Dead-Zone & System Performance

As mentioned in Section 3.2.6.1 the dead-zone nonlinearity effects impacts the control errors as well as fuel usage. Previously, the dead-zone has been set to 0.0002

to minimize control errors. Although this decision does not have a major impact within this thesis, future applications of this control algorithm may require a different balance between fuel consumption and accuracy. Therefore, a dead-zone trade study was performed to illustrate the relationship between control errors and fuel consumption for a given dead-zone. This allows users to select which dead-zone is best for their particular application of the speed and path control algorithm developed herein. Figure 4.13 describes the relationship between the control error and the fuel consumption.

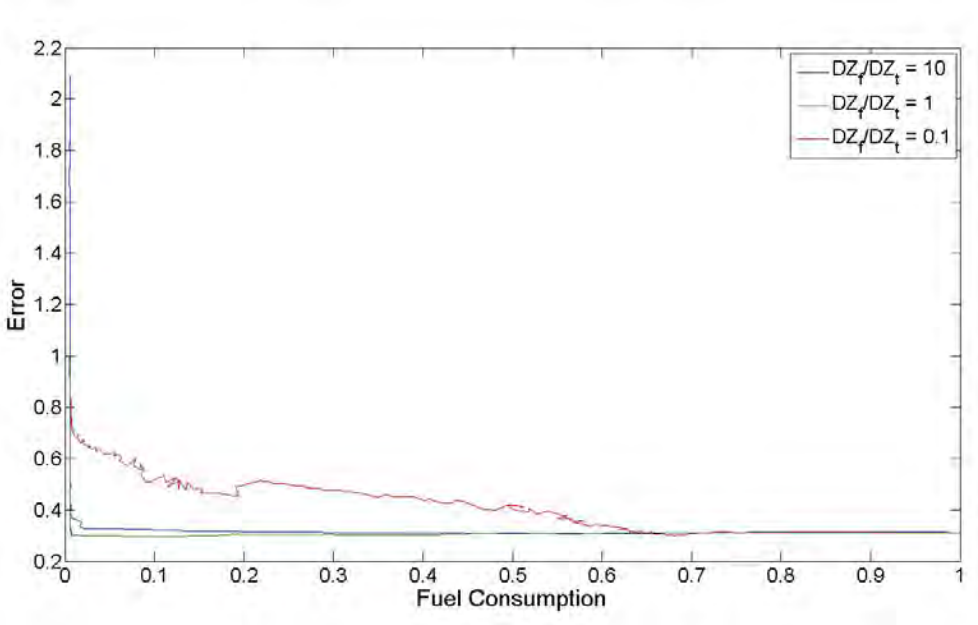

Figure 4.13: Relationship Between Control Error & Fuel Consumption

Figure 4.13 demonstrates the relationship between the control error and fuel consumption. The error term considers errors in both the satellites position and orientation. Since these units are not equal, the author assigned one unit of error equal to one centimeter or position error and half a degree of pointing error. In addition, since this is a relative comparison of error, the errors are then normalized so that the max error when the dead-zone ratios are equal is one. Additionally, DZ_f/DZ_t is the ratio between the dead-zone values applied to control signal for force and the control signal for torque respectively. A smaller dead-zone improves accuracy at the expense

of fuel, and the reverse is true when the dead-zone is relatively large. While this is to be expected, this plot also shows that setting the dead-zone values differently results in worse performance. This is attributed to the fact that a satellite's orientation and position in the global frame are still coupled even though the thruster pattern to determine a satellite's position and velocity is not. Next, Figure 4.14 illustrates how error and fuel usage varies with dead-zone.

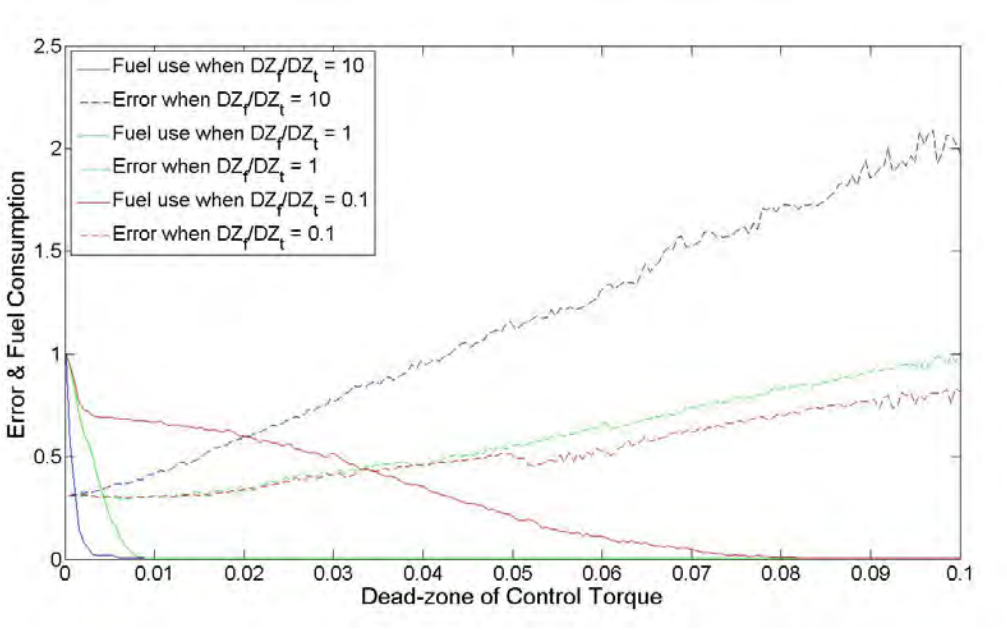

Figure 4.14: Comparison of Dead-zone with Control Error & Fuel Consumption

Figure 4.14 further reinforces the indirect relationship between fuel consumption and control error. Furthermore, this plot provides further explanation to why the dead-zone values should be the same for both the force and torque signals. For this test, the translational errors require more control to direct as necessary than the rotational error. Thus, when the dead-zone applied to the control force is larger than the dead-zone applied to the control torque, greater errors are manifested in the simulation. At the same time however, the fuel savings improve when compared to the nominal case of $DZ_f/DZ_t = 1$. Since the relative difference in the errors between the two cases far outweighs the relative savings in fuel consumption, the red $DZ_f/DZ_t = 10$ line of Figure 4.13 becomes worse. On the contrary, when $DZ_f/DZ_t = 0.1$ the

control force passes through a much smaller dead-zone than the control torque. This results is a smaller error than the other cases, but the resulting fuel cost to do so is much greater. Thus, $DZ_f/DZ_t = 1$ provides the best relationship for the dead-zones. In addition, for this particular path, one would select a dead-zone of 0.01 for both the dead-zone nonlinearities if one was more interested in fuel efficiency than control error.

4.5 Summary of Research Results

The speed and path control algorithm has been validated through the use of an inspection maneuver. Furthermore, this control algorithm is capable of keeping a satellite's position error to within two millimeters, its velocity error to within one millimeter per second, and its pointing error to within two degrees. This has been demonstrated through theoretical simulation with experimentally derived hardware values. Nonetheless, in order to achieve the same level of precision on the actual SPHERES platform the gains optimized for the theoretical simulation may need to be tweaked to ensure the control algorithm is optimized when the satellite's realisms are included. The author believes this can be done by performing the same method for gain optimization as was performed for each of the gains in Chapter III. Additionally, the dead-zone investigation discussed in Section 4.4 illustrates how future users can select the dead-zone value for each controller to meet their mission requirements. Again, this test only considered theoretical conditions for the described inspection maneuver. As a result, it is possible that these specific dead-zone values may not reflect the user's desires when tested in reality. Thus, the code for the dead-zone study is included in Appendix D so that future users may use the same process for investigating the dead-zone values affect accuracy and fuel efficiency. The bottom line is that when applying this control algorithm to the actual SPHERES program, the methods for determining the specific gain and dead-zone values should be of more importance than the actual values provided within this research. Nonetheless, this research has made a few contributions to the SPHERES program and opened up

opportunities for future work as well. These topics are the discussion of the next chapter.

V. Conclusions

The goal of this research was to investigate enhancements to the SPHERES software control suite and provide MIT's SPHERES program with a speed and path control algorithm. The speed and path control algorithm produced within this thesis is capable of commanding SPHERES to meet user-specified positions, orientations, and velocities along relative paths within required tolerances. Specifically, the controller is capable of allowing an inspector to track a target to within two degrees while translating along a path with less than two millimeters of position error and a millimeter per second of velocity error. Once implemented, this control algorithm will enable the SPHERES program to further pursue research on in-space robotic assembly and other pursuits in which velocity control is particularly advantageous.

5.1 Research Contributions

Although work within the realm of relative satellite motion and formation spaceflight has been extensive, this research has contributed to the field. Specifically, this research has introduced a control algorithm capable of being used for applications involving precise speed and path control. In addition to creating a speed and path controller for MIT's SPHERES satellites, a simulation for SPHERES was created without requiring the use of C++ code. Thus, this simulation contributes to MIT's SPHERES program by allowing future guest scientists of the SPHERES program to interact with SPHERES without knowledge of C++. This capability allows a wider pool of control engineers to provide insight into this program. With this in mind, the MATLAB® master script and the SIMULINK® simulation it runs have been included in Appendices A and B (and will be made available) for use as a basis for future guest scientists to start from.

A method for selecting the gains for this control algorithm has also been supplied. As missions and desire evolve, the specific values of the gains used within this thesis may not reflect the optimal gains for future requirements. The method of gain selection used within this thesis however, can still be applied to determine the optimal

gains for different program requirements. The MATLAB® scripts used to determine the optimal gains for the translational and rotational controllers are supplied in Appendix C. In addition, mission requirements may dictate that a different dead-zone value should be selected to produce the desirable relationship between accuracy and fuel efficiency. Figure 4.14 is provided to give users an understanding of how the selection of the dead-zones affects control error and fuel consumption. Accompanying this chart, the MATLAB® code used to perform the dead-zone trade study is also provided in Appendix D should further consideration be required.

5.2 Recommendations for Future Work

Up to this point the control algorithm has been developed, tested, and shown to perform within reasonable tolerances. Yet this research only lays the groundwork for the algorithm to be used on-board the satellites of the SPHERES program. The next step is to convert the control algorithm into C++, and test the algorithm on the SPHERES at MIT in their Space Systems Laboratory. Upon conclusion of this, the control algorithm should be used on-board the International Space Station for further analysis and application. At this stage the control algorithm would be used for MIT's SPHERES program to continue to provide a practical intermediate step to develop, test, and validate autonomous formation spaceflight algorithms.

Finally, other avenues exist to further pursue research on this topic. Among them includes incorporating the option for added realism within this simulation. This could be achieved by including attitude and position error with noise corrupted measurements, a changing mass moment of inertia from fuel consumption, or including affects of air drag from within the ISS. An estimator would also need to be added to this simulation as well. Another approach worth pursuing includes developing a better user interface to allow parameterized paths or velocities to be input in a more efficient manner. Optimal paths could also be developed that would take advantage of the unique abilities of a speed and path controller. Subsequent research could also be performed to include the option for SPHERES to switch between desired targets

during a simulation once an inspection or other maneuver is completed. This would further enhance the development of using satellites for in-space refueling or robotic assembly.

Appendix A. Algorithm Script

The following MATLAB® script is used to operate the SPHERES simulation used in this thesis. It should be noted that additional subroutines executed within the script are included below the master script.

A.1 Simulation Master Script

The following code is titled 'SPHERES_simulation.m' and is included below for reference within this thesis

Listing A.1: Appendix1/SPHERES.m

```
%%%%%%%%%%%%%%%%%%%%%%%% SPHERES SIMULATION %%%%%%%%%%%%%%%%%%%%%%%%
%
% Master Script to model speed & path algorithms for MIT's SPHERES
% program
%
% Author: Sam Barbaro       AFIT ENY-3                  06 Jan 2012
%
% Purpose: This script feeds constants & variables into the
% simulation SPHERES_3D_Simulation.mdl for various paths that can
% be set by the user.  This script then plots the comparison of
% how well the states actually met the desired values.
%
% Model: This simulation considers all thrusters number 0-11, and
% is capable of changing SPHERES orientation as well as its
% position in global space.
%
% Programs Called: SPHERES_3D_Simulation_v3.mdl, skew.m,
%                  datainterp.m
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc; clear all; close all;
```

```
%% Plan Desired Path and Speeds for SPHERES
%% & Set Initial Conditions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% USER MAKES CHANGES HERE %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Set Simulation Parameters
StepSize = 0.005;     % How often should simulation update [sec]
Duration = 260.0;     % How long to run simulation [sec]
% Set Deadzone Bandwidth of Control Force & Control Torque
Dband_f  = 0.0002;    % Deadzone Bandwidth of Control Force
Dband_t  = 0.0002;    % Deadzone Bandwidth of Control Torque
% Set Rate Limits
Rate_T   = .10;       % S/C will not exceed this speed     [m/s]
Rate_R   = 6;         % S/C will not spin faster than this [deg/s]

% Define Path of SPHERES (The Target)
% This is done by picking desired relative positions and
% velocities in the global frame, as well as the Euler angles at
% each time of interest. There is no limit to how many times you
% select but a position, velocity, & Euler angle must be assigned
% for each time you select.  In addition the Euler angles are for
% a Roll,Pitch,Yaw configuration like many aircraft. Lastly, an
% "c" is affixed to the beginning of these variables to denote
% that this information is for the SPHERES that defines the origin
% of the relative frame.  This SPHERES is the chief to be
% inspected
    % A POINT MUST BE SPECIFIED AT BEGINING AND END OF SIMULATION
    % example: Note that time is in units of seconds, position and
    % velocity are in the global frame and in meters and meters
    % per second respectively, and Euler Angles are in degrees.
    % ex: time = sec; pos = [x y z]; EA = [r p y]
    % ex: t(#,1)=0.5; p(#,1:3)=[1 0.02 0.5];
        ct(1,1) = 0.0;            cp(1,1:3) = [0.00  0.00 0.00];
        ct(3,1) = Duration;       cp(3,1:3) = [0.50  1.00 2.00];
```

```
%       ct(1,1)  = 0.0;       cp(1,1:3)  =[ 0  2.0000       0];
%       ct(2,1)  = 13.7;      cp(2,1:3)  =[ 0  1.9984  0.0789];
%       ct(3,1)  = 27.4;      cp(3,1:3)  =[ 0  1.9938  0.1577];
%       ct(4,1)  = 41.0;      cp(4,1:3)  =[ 0  1.9860  0.2363];
%       ct(5,1)  = 54.7;      cp(5,1:3)  =[ 0  1.9751  0.3145];
%       ct(6,1)  = 68.4;      cp(6,1:3)  =[ 0  1.9612  0.3922];
%       ct(7,1)  = 82.1;      cp(7,1:3)  =[ 0  1.9442  0.4693];
%       ct(8,1)  = 95.8;      cp(8,1:3)  =[ 0  1.9241  0.5456];
%       ct(9,1)  = 109.5;     cp(9,1:3)  =[ 0  1.9011  0.6211];
%       ct(10,1) = 123.2;     cp(10,1:3) =[ 0  1.8751  0.6957];
%       ct(11,1) = 136.8;     cp(11,1:3) =[ 0  1.8462  0.7691];
%       ct(12,1) = 150.5;     cp(12,1:3) =[ 0  1.8144  0.8414];
%       ct(13,1) = 164.2;     cp(13,1:3) =[ 0  1.7798  0.9123];
%       ct(14,1) = 177.9;     cp(14,1:3) =[ 0  1.7424  0.9819];
%       ct(15,1) = 191.6;     cp(15,1:3) =[ 0  1.7023  1.0499];
%       ct(16,1) = 205.3;     cp(16,1:3) =[ 0  1.6595  1.1162];
%       ct(17,1) = 218.9;     cp(17,1:3) =[ 0  1.6142  1.1808];
%       ct(18,1) = 232.6;     cp(18,1:3) =[ 0  1.5663  1.2436];
%       ct(19,1) = 246.3;     cp(19,1:3) =[ 0  1.5160  1.3045];
%       ct(20,1) = Duration;  cp(20,1:3) =[ 0  1.4634  1.3633];

    % Set SPHERES Initial Rates
  Euler_c  = [0;0;0];   % initial euler angles        [deg]
  Omega_c  = [0;0;0];   % initial angular rates       [deg/s]
  Pos_c    = cp(1,1:3)';% initial position of SPHERES [m]   (global)
  Vel_c    = [0;0;0];   % initial velocity of SPHERES [m/s] (global)
  % Set SPHERES Target Pointing Vector
  Point_c  = [0;0;-1];   % desired pointing vector     [m]   (body)
                         % body frame vector specifying what part of
                         % the inspector faces the target.  This ...
                            could
                         % be for a camera or docking mechanism

% Define Path of SPHERES (The Inspector)
```

```
% This is done by picking the inspector's desired range from the
% target in the global frame.  In addition, the inspector's
% initial conditions and desired pointing vector must also be
% specified.  The desired position is specified as how far the
% inspector is away from the target using the global frames
% coordinates.  The initial conditions specify where the satellite
% is in the global frame.  Lastly, the desired pointing vector is
% a (3x1) body frame vector which identifies which part of the
% inspector is to point to the target.  Finally, an "i" is affixed
% to the beggining of these variables to denote that this
% information is for the SPHERES that inspects the other(s)
    % example: Note that time is in units of seconds, position is
    % in the relative frame and in meters.  The physical units of
    % the pointing vector are irrelevant as the vector will be
    % normalized.  Just make sure the vector has the same units
    % in each component.
    % ex: time = sec; pos = [x y z]; pv = [x y z]
    % ex: t(#,1)=0.5; p(#,1:3)=[1 0.02 0.5];

% it(1,1)  = 0.0;      ip(1,1:3)  = [ -0.7000   0.0000   0.0000];
% it(2,1)  = Duration; ip(2,1:3)  = [ -0.7000   0.0000   0.0000];


it(1,1)  = 0.0;      ip(1,1:3)  = [ 0.4000   0.0000  -0.2000];
it(2,1)  = 10;       ip(2,1:3)  = [ 0.3864   0.1035  -0.1778];
it(3,1)  = 25;       ip(3,1:3)  = [ 0.3464   0.2000  -0.1556];
it(4,1)  = 39;       ip(4,1:3)  = [ 0.2828   0.2828  -0.1333];
it(5,1)  = 54;       ip(5,1:3)  = [ 0.2000   0.3464  -0.1111];
it(6,1)  = 69;       ip(6,1:3)  = [ 0.1035   0.3864  -0.0889];
it(7,1)  = 84;       ip(7,1:3)  = [ 0.0000   0.4000  -0.0667];
it(8,1)  = 98;       ip(8,1:3)  = [-0.1035   0.3864  -0.0444];
it(9,1)  = 113;      ip(9,1:3)  = [-0.2000   0.3464  -0.0222];
it(10,1) = 128;      ip(10,1:3) = [-0.2828   0.2828   0.0000];
it(11,1) = 142;      ip(11,1:3) = [-0.3759   0.1368   0.0222];
it(12,1) = 154;      ip(12,1:3) = [-0.4000   0.0000   0.0444];
```

```
    it(13,1) = 172;       ip(13,1:3) = [-0.3464   -0.2000    0.0667];
    it(14,1) = 186;       ip(14,1:3) = [-0.2294   -0.3277    0.0889];
    it(15,1) = 201;       ip(15,1:3) = [-0.0695   -0.3939    0.1111];
131 it(16,1) = 216;       ip(16,1:3) = [ 0.1035   -0.3864    0.1333];
    it(17,1) = 231;       ip(17,1:3) = [ 0.2571   -0.3064    0.1556];
    it(18,1) = 245;       ip(18,1:3) = [ 0.3625   -0.1690    0.1778];
    it(19,1) = 260;       ip(19,1:3) = [ 0.4000    0.0000    0.2000];
    it(20,1) = Duration; ip(20,1:3) = [ 0.4000    0.0000    0.2000];
136
        % Set SPHERES Initial Rates
    Euler_i   = [0;0;0];    % initial euler angles         [deg]
                            % [roll, pitch, yaw] 3-2-1 Rotation
    Omega_i   = [0;0;0];    % initial angular rates        [deg/s]
141 Pos_i     = ip(1,1:3)'+Pos_c;
                            % initial position of SPHERES [m]   (global)
    Vel_i     = [0;0;0]+Vel_c;
                            % initial velocity of SPHERES [m/s] (global)

146     % Set SPHERES Inspector Pointing Vector
    Point_i   = [0;0;1];    % desired pointing vector      [m]   (body)
                            % body frame vector specifying what part of
                            % the inspector faces the target. This could
                            % be for a camera or docking mechanism
151
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%% END TYPICAL USER CHANGES %%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

156 %% Load SPHERES Constants
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Mass Moments of Inertia
    Eye          = [1 0 0; 0 1 0; 0 0 1];   % Identity Matrix
    MOI_Wet = [ 2.30e-2    9.90e-5   -2.95e-4;% relative to COM [kg*m^2]
161              9.90e-5    2.42e-2   -2.54e-5;% SPHERES MOI w/ full tank
                -2.95e-4   -2.54e-5    2.14e-2;];
```

```
MOI_Dry = [ 2.19e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
            9.90e-5   2.31e-2  -2.54e-5;% SPHERES MOI w/empty tank
           -2.95e-4  -2.54e-5   2.13e-2;];
InMOI_Wet    = inv(MOI_Wet);            % shorthand
InMOI_Dry    = inv(MOI_Dry);            % shorthand
OneD_MOI_Wet = MOI_Wet(2,2);            % MOI used for 1-D example
OneD_MOI_Dry = MOI_Dry(2,2);            % MOI (dry) used for 1-D

% Mass of SPHERES
Mass_Wet  = 4.16;       % SPHERES mass with fuel in [Kg]
Mass_Dry  = 3.55;       % SPHERES mass without fuel in [Kg]
% Thruster Information
F   = 0.11;             % force of individual thruster [N]
l   = 0.193;            % length between thrusters [m]
% SPHERES Plant
A   = [0 0 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0 0 1;...
       0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0;];
Bu  = [0 0 0; 0 0 0; 0 0 0; ...
      1/Mass_Wet 0 0; 0 1/Mass_Wet 0; 0 0 1/Mass_Wet];
C   = eye(6);           % assumes all states provided by estimator
D   = zeros(6,3);       % control does not directly affect output
% Path & Speed Controller Gain Information
Kd  = 0.352;            % Gain for Quaternion Controller
Q1  = 1;                % "Q" Weight for Position in LQR Controller
Q2  = 0.018;            % "Q" Weight for Velocity in LQR Controller
R   = 0.06;             % "R" Weight for LQR Controller
Q   = blkdiag(Q1,Q1,Q1,Q2,Q2,Q2); % "Q" Weighting Matrix
R   = blkdiag(R,R,R);   % "R" Weighting Matrix
[K,~,~] = lqr(A,Bu,Q,R);% LQR gain
% Preditiction Term
tau = 2;                % determine slope of switch line (M=-1/tau)
% Define Dead Zone Range
                        % these values ensure translation stays
                        % w/in specified bandwidth
High_f = Dband_f/2;     % force values above this aren't reset
```

```
    Low_f   = -Dband_f/2;    % force values below this aren't reset
    High_t  = Dband_t/2;     % torque values above this aren't reset
    Low_t   = -Dband_t/2;    % torque values below this aren't reset
% Update Rate Limit Value
    Rate_R = deg2rad(Rate_R);% convert from degrees to radians
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %% Run Simulation
% run simulation for target SPHERES and then manipulate data to
    % update information for the other SPHERES to track.

    %%%%%%%%%%%%%%%%%%%%%%%%% SPHERES Chief %%%%%%%%%%%%%%%%%%%%%%%%%
        % Paramaterize Important Points in Time
[time,pos_c,velo,quat] = datainterp_v1(StepSize,Duration,ct,cp,...
       Euler_c);
        % Convert Global Frame Info into Body Frame Info
    Quat_0      = quat;          % Find Initial Quaternion Vector (4x1)
    slpint      = pos_c(1,1:3)';% Initial Slope Intercept of Position
    Pos_0       = Pos_c;         % Call Position in Global Frame  (3x1)
Vel_0       = Vel_c;         % Call Velocity in Global Frame  (3x1)
    Omega_0     = deg2rad(Omega_c);% Convert to [rad/s]          (3x1)
    Initial     = [Pos_0; Vel_0];% Simulation input for State Space
    Point       = Point_c;       % Desired pointing vector (target s/c)
        % Determine Where S/C should point
Targ = [pos_c(:,1), pos_c(:,2), pos_c(:,3)-ones(length(time),1)];
                                 % Target will point in same direction
                                 % of global frame for entire simulation
        % Load Desired Data into Look-Up Tables (Global Frame)
    Des_Vel_X   = velo(:,1);     % Desired Velocity on X-axis
Des_Vel_Y   = velo(:,2);     % Desired Velocity on Y-axis
    Des_Vel_Z   = velo(:,3);     % Desired Velocity on Z-axis
    Targ_Pos_X = Targ(:,1);      % Target s/c points down
    Targ_Pos_Y = Targ(:,2);      % Target s/c points down
    Targ_Pos_Z = Targ(:,3);      % Target s/c points down
    % Define Mass Moment of Inertia Matrix
```

```
                                % Eventually it would be nice to
                                % include a variable MOI matrix that
                                % changes as the fuel is consumed,
                                % however due to complexity, that has
236                             % not been accounted for at this point
                                % in the design
    % MOI       = 2e-2*Eye;     % override to principal axis for test
    % IMOI      = inv(MOI);
    MOI       = MOI_Wet;        % MMOT when SPHERES' fuel tank is full
241 IMOI      = InMOI_Wet;      % Inverse of MMOI of SPHERES when full
        % Run Simulation
    sim('SPHERES_3D_Simulation_v3');
        % Save Values
    Position_Ch_d     = Desired_Values.signals.values(:,1:3);
246 Velocity_Ch_d     = Desired_Values.signals.values(:,4:6);
    Position_Ch_e     = Errors.signals.values(:,1:3);
    Velocity_Ch_e     = Errors.signals.values(:,4:6);
    Quaternion_Ch_e   = Errors.signals.values(:,7:10);
    Errors_Ch         = Errors.signals.values(:,1:11);
251 Position_Ch_bf    = States.signals.values(:,1:3);
    Velocity_Ch_bf    = States.signals.values(:,4:6);
    Quaternion_Ch     = States.signals.values(:,7:10);
    EulerRate_Ch      = States.signals.values(:,11:13);
    Position_Ch_gf    = States_GlobalFrame.signals.values(:,1:3);
256 Velocity_Ch_gf    = States_GlobalFrame.signals.values(:,4:6);
    Control_For_Ch    = Control.signals.values(:,1:3);
    Control_Tor_Ch    = Control.signals.values(:,4:6);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
261
    %%%%%%%%%%%%%%%%%%%% SPHERES Inspector %%%%%%%%%%%%%%%%%%%%%%%%%%
        % Paramaterize Important Points in Time
        % This is done for two vectors.  First the vector specifing
        % where the desired position with respect to the chief is
266     % interpolated for comparision in plots.  Second the vector
```

```
        % specifing where the desired position is with respect to the
        % global frame is propogated for use within the simulation.
    [time,pos_i,velo,quat] = datainterp_v1(StepSize,Duration,it,ip,...
       Euler_i);
    pos  = pos_i + Position_Ch_gf;
271 velo = velo + Velocity_Ch_gf;
        % Convert Global Frame Info into Body Frame Info
    Quat_0 = quat;              % Find Initial Quaternion Vector (4x1)
    slpint   = pos (1,1:3)';    % Initial Slope Intercept of Position
    Pos_0    = Pos_i;           % Call Position in Global Frame  (3x1)
276 Vel_0    = Vel_i;           % Call Velocity in Global Frame  (3x1)
    Omega_0  = deg2rad(Omega_i);% Convert to [rad/s]            (3x1)
    Initial  = [Pos_0; Vel_0]; % Simulation input for State Space
    Point      = Point_i;       % Pointing vector      (inspector s/c)
        % Load Desired Data into Look-Up Tables (Global Frame)
281 Des_Vel_X= velo(:,1);       % Desired Velocity on X-axis
    Des_Vel_Y= velo(:,2);       % Desired Velocity on Y-axis
    Des_Vel_Z= velo(:,3);       % Desired Velocity on Z-axis
    Targ_Pos_X = Position_Ch_gf(:,1); % s/c should point to target
    Targ_Pos_Y = Position_Ch_gf(:,2); % s/c should point to target
286 Targ_Pos_Z = Position_Ch_gf(:,3); % s/c should point to target
        % Define Mass Moment of Inertia Matrix
                                % Eventually it would be nice to
                                % include a variable MOI matrix that
                                % changes as the fuel is consumed,
291                             % however due to complexity, that has
                                % not been accounted for at this point
                                % in the design
    % MOI       = 2e-2*Eye;     % override to principal axis for test
    % IMOI      = inv(MOI);
296 % MOI       = MOI_Wet;      % MMOT when SPHERES' fuel tank is full
    % IMOI      = InMOI_Wet;    % Inverse of MMOI of SPHERES when full
        % Run Simulation
    sim('SPHERES_3D_Simulation_v3');
        % Save Values
```

```
Position_In_d    = Desired_Values.signals.values(:,1:3);
Velocity_In_d    = Desired_Values.signals.values(:,4:6);
Position_In_e    = Errors.signals.values(:,1:3);
Velocity_In_e    = Errors.signals.values(:,4:6);
Quaternion_In_e  = Errors.signals.values(:,7:10);
Errors_In        = Errors.signals.values(:,1:11);
Position_In_bf   = States.signals.values(:,1:3);
Velocity_In_bf   = States.signals.values(:,4:6);
Quaternion_In    = States.signals.values(:,7:10);
EulerRate_In     = States.signals.values(:,11:13);
Position_In_gf   = States_GlobalFrame.signals.values(:,1:3);
Velocity_In_gf   = States_GlobalFrame.signals.values(:,4:6);
Control_For_In   = Control.signals.values(:,1:3);
Control_Tor_In   = Control.signals.values(:,4:6);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Relative Information
    % Find distance between true Chief and desired Inspector
       % Distance = Position_InspectorDesired - Position_TrueChief
    RHO_des = pos  - Position_Ch_gf;
    VEL_des = velo - Velocity_Ch_gf;
    % Find distance between true Chief and true Inspector\
       % Distance = Position_TrueInspector - Position_TrueChief
    RHO_tru = Position_In_gf - Position_Ch_gf;
    VEL_tru = Velocity_In_gf - Velocity_Ch_gf;
    % Shift user input points into correct location
    IP = zeros(size(ip,1),3);
for ctr = 1:size(ip,1)
    ind = find(time ≥ it(ctr),1,'first');
    IP(ctr,1:3) = RHO_des(ind,1:3);
end

%% Format and Plot Results
```

```
% Plotting and Animation Commands are removed for script included
% within the thesis because plotting commands don't add to the
% focus of this research
```

A.2 Data Interpretation

Within the 'SPHERES_Simulation.m' code the sub-routine 'datainterp.m' is called to interpret the user inputs for each SPHERES in the simulation. The function is included below for reference.

Listing A.2: Appendix1/datainterp.m

```
function [time,pos,velo,quat]=datainterp_v1(SimDelT,SimTime,tpt,...
    ppt,euler)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Author: Sam Barbaro            AFIT ENY-3            10 Jan 2012
%
% Purpose: This function is a sub-routine for the SPHERES
% simulation.  It serves to take users desired values at each
% point of interest and develop arrays that can be built into a
% table in the actual simulation.
%
% Inputs:
%  SimDelT- Time Step Used in Simulation     (nx1) Vector [sec]
%  SimTime- Total Time Simulation is Run     (1x1) Scalar [sec]
%  tpt-     Time Points Specified by User    (mx1) Vector [sec]
%  ppt-     Posistion Points Chosen by User  (mx3) Vector [m]
%  euler-   Initial Rotation by Euler Angle  (3x1) Vector [deg]
%
% Outputs:
%  time-    Time Array                       (nx1) Vector [sec]
%  pos-     Position Array                   (nx3) Vector [m]
%  velo-    Velocity Array                   (nx3) Vector [m/sec]
```

```
%  quat-    Initial Quaternion Vector        (4x1) Vector []
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Find Time Array
time   = 0:SimDelT:SimTime;     % build time array for simulation
time   = time';                 % make time a column vector

%% Initialize Variables
len    = length(tpt);           % find number of points of interest
dim    = length(time);          % find length of time array
ind    = zeros(len,1);          % initialise indexing term for loop
mx     = zeros(dim-1,1);        % initialize slope of x vs. t
my     = mx;                    % initialize slope of y vs. t
mz     = mx;                    % initialize slope of z vs. t
bx     = mx;                    % initialize x-intercept
by     = mx;                    % initialize y-intercept
bz     = mx;                    % initialize z-intercept
velo   = zeros(dim,3);          % initialize desired velocity vector
pos    = velo;                  % initialize desired position vector

%% Find Slope and Intercept Information
for ctr = 1:len
  ind(ctr) = find(time ≥ tpt(ctr),1,'first');
  pos(ind(ctr),:)   = ppt(ctr,1:3);
  if ctr ≤ len-1
    mx(ctr,1)=(ppt(ctr+1,1)-ppt(ctr,1))/(tpt(ctr+1,1)-tpt(ctr,1));
    my(ctr,1)=(ppt(ctr+1,2)-ppt(ctr,2))/(tpt(ctr+1,1)-tpt(ctr,1));
    mz(ctr,1)=(ppt(ctr+1,3)-ppt(ctr,3))/(tpt(ctr+1,1)-tpt(ctr,1));
    bx(ctr,1)= ppt(ctr,1)-mx(ctr,1)*tpt(ctr,1);
    by(ctr,1)= ppt(ctr,2)-my(ctr,1)*tpt(ctr,1);
    bz(ctr,1)= ppt(ctr,3)-mz(ctr,1)*tpt(ctr,1);
  end
end
```

```
%% Build Position & Velocity Vector
    % If position   = slope*time+intercept
    % Then velocity = slope
for block = 1:len-1;
    for ctr = ind(block,1):1:ind(block+1,1);
        pos(ctr,:)  = [mx(block)*time(ctr,1)+bx(block), ...
                       my(block)*time(ctr,1)+by(block), ...
                       mz(block)*time(ctr,1)+bz(block)];
        velo(ctr,:) = [mx(block) my(block) mz(block)];
    end
end

%% Find Inital Quaternions
    % verify Euler Angles values are between 0 & 360 degrees
euler = mod(euler,360);
    % convert Euler Angles to radians
euler = deg2rad(euler);

    % Convert Euler Angles to rotation matrix
se1 = sin(euler(1)); ce1 = cos(euler(1));
se2 = sin(euler(2)); ce2 = cos(euler(2));
se3 = sin(euler(3)); ce3 = cos(euler(3));
    % calculate 3-2-1 rotation matrix based of euler angles
R   = [ ce2*ce3   ce1*se3 + se1*se2*ce3   se1*se3 - ce1*ce3*se2;...
       -ce2*se3   ce1*ce3 - se1*se2*se3   se1*ce3 + ce1*se3*se2;...
          se2         -se1*ce2                 ce1*ce2];
    % find quaternion vector (4x1) from rotation matrix
    % (use eigen-axis)
tr=trace(R);
if abs(tr-3) ≤ eps                  % no rotation
    a = [0 0 1];                     %   eigen-axis
    phi = 0;                         %   euler angle
    q = [0;0;0;1];                   %   quaternion vector
else                                 % arbitary rotation
    phi=acos((1/2)*(tr-1));          %   euler angle
```

```
92      ax=(1/(2*sin(phi)))*(R'-R);      %   skewed representation
        a=[ax(3,2);ax(1,3);ax(2,1)];     %   of a eigen-axis
        q4=cos(phi/2);                   %   4th quaternion
        qu=sin(phi/2)*a;                 %   1st 3 quaternions
        q=[qu;q4];                       %   combine quaternions
97  end
    quat = q;                            % save quaternion
```

A.3 Skew Matrix

Within the 'SPHERES_Simulation.m' code the sub-routine 'skew.m' is called to convert a three by one column vector into that vector's skew representation. The function is included below for reference.

Listing A.3: Appendix1/skew.m

```
 1  function x_cross=skew(x)
    %Converts a 3 by 1 vector into a skew-symmetric matrix

    % Check for correct size

 6  if max(size(x))≠3 || min(size(x))≠ 1
        disp('not a 3by1 vector')
        return
    end

11  x_cross = [0 -x(3) x(2); x(3) 0 -x(1); -x(2) x(1) 0];

    end
    %eof
```

Appendix B. Simulation Diagrams

The SIMULINK® diagram used to develop and test the speed and path control algorithm described within this thesis has been included here for reference. This simulation consists of numerous subsystems. Thus the appendix is organized to show the highest level of detail first, and then discuss each section or 'block' with their subsystems. This appendix is ordered to discussed the 'Error Determination' block in Section B.1, the 'Speed & Path Controller' block next in Section B.2, followed by the 'SPHERES Plant' block in Section B.3, and lastly, the 'User Inputs' and 'Outputs' blocks are discussed in Section B.4. The SIMULINK® overview is shown in Figure B.1. This picture depicts how the top-level of the SIMULINK® model appears.

B.1 Error Determination

The 'Error Determination' block performs all functions described in Section3.1. The block's overview is shown illustrated in Figure B.2. This block has three subsystems. The first subsystem splits the satellite's state vector and is depicted through Figure B.3. Once the translation error has been determined in the global frame, Figure B.6 shows how the next subsystem rotates those errors to the body frame of the satellite. Figure B.7 shows how the quaternion error is calculated to determine how to rotate the satellite body frame to the desired orientation.

B.1.1 Split State Subsystem. Figure B.3 shows how the states are split into four main components. One section contains the position vector, one holds the velocity vector, a third contains the quaternions, and the fourth component is the angular rates of the satellite.

The 'Split-States' block not only divides the state vector into the four groups, but it also uses the quaternions to compute the rotation matrix that converts information out of the global frame and into the body frame. This process is expounded upon in Figure B.6.

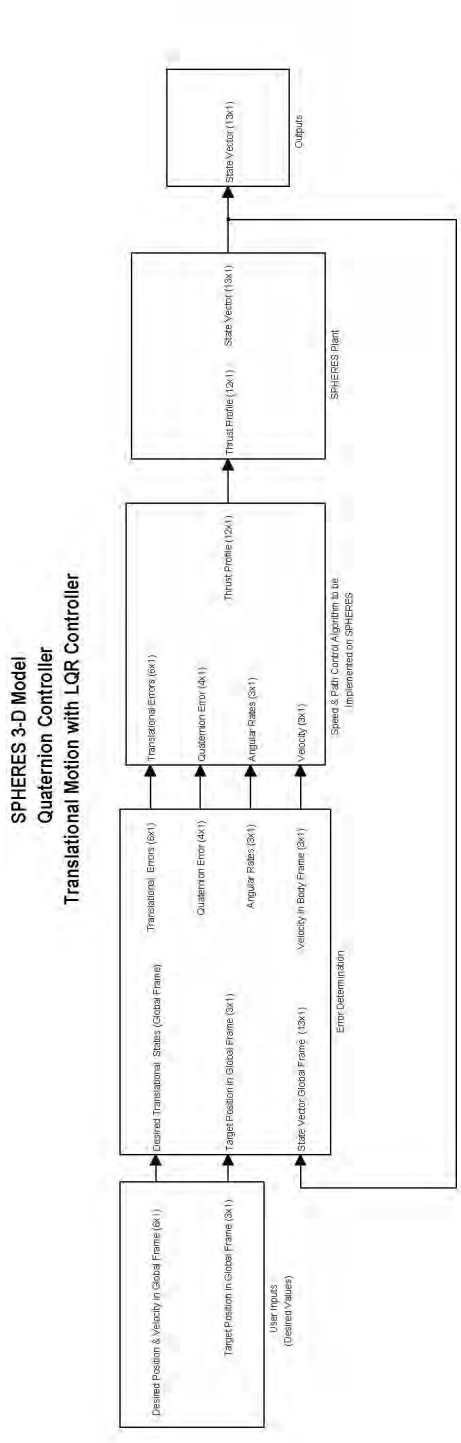

Figure B.1: Speed & Path Control Simulation Overview

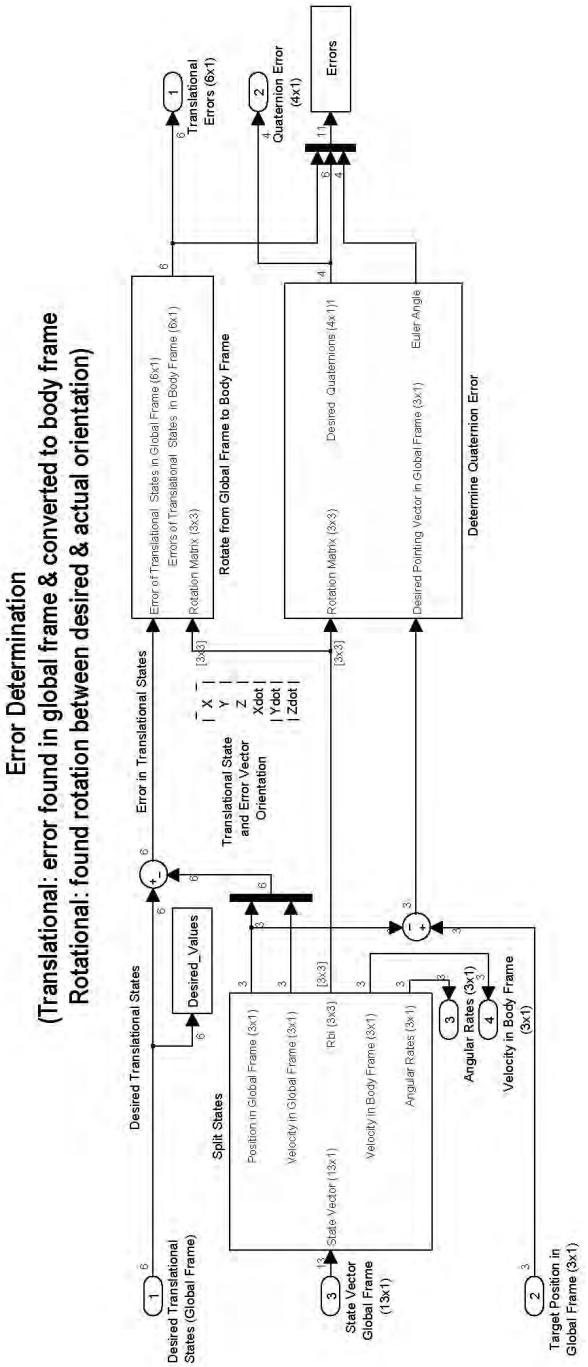

Figure B.2: Error Determination Overview

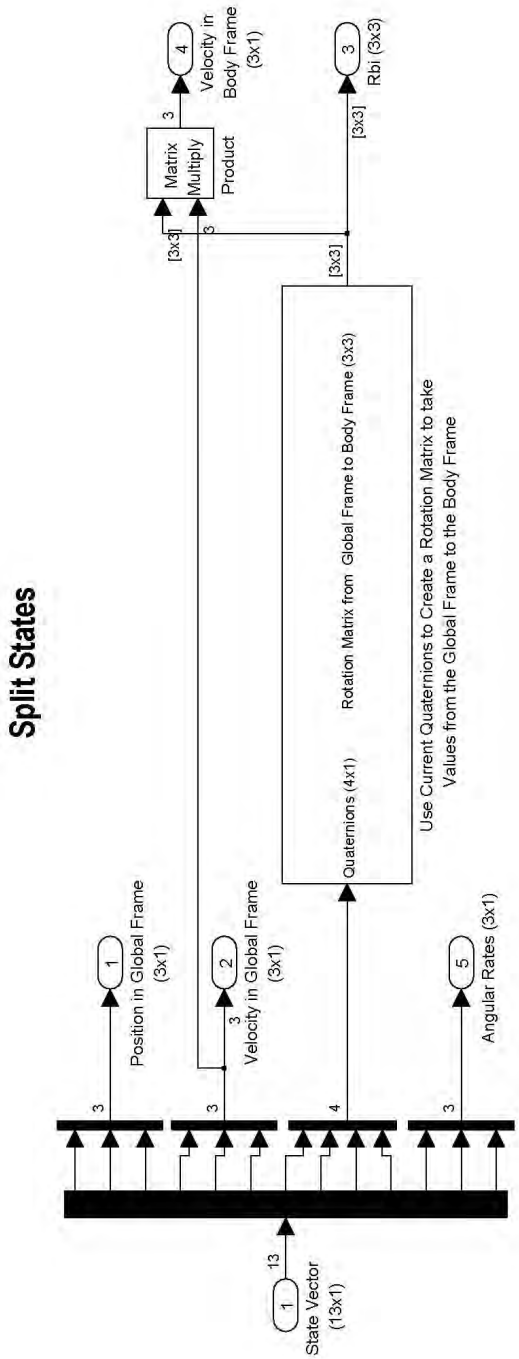

Figure B.3: Split State Vector

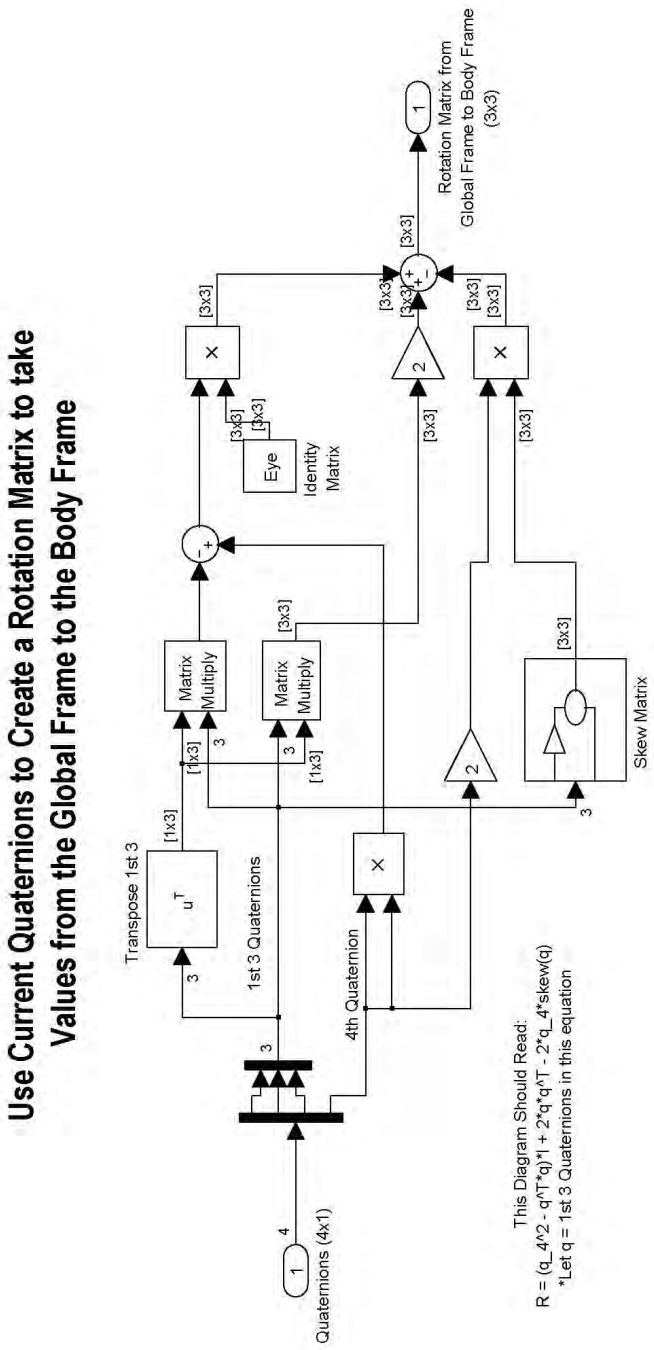

Figure B.4: Rotation Matrix to Rotate Information from Global Frame to Body Frame

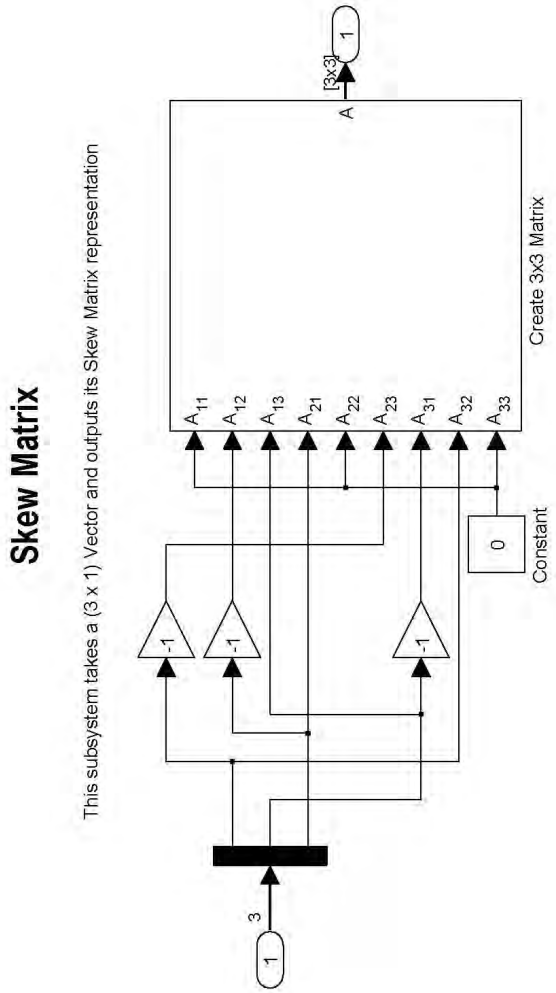

Figure B.5: Skew Matrix

B.1.2 Subsystem to Determine Translational Errors. The translational errors are determined in the global frame, but must be rotated into the body frame before they can be used by the controller. Figure B.6 describes how this process is performed.

B.1.3 Subsystem to Determine Quaternion Error. The quaternion error needs to be calculated before the satellite orientation can be restored. This is done by determining where the satellite is pointing and where it should point. Next the eigenaxis and principle Euler angle are determined from these two vectors. The desired quaternions can then be converted from the eigenaxis and Euler angle. The singularity from the eigenaxis method is thus avoided with the use of an 'if/else'. Figure B.7 shows how the desired quaternions are created and Figure B.8 depicts how the eigenaxis and Euler angle are found.

B.2 Speed & Path Controller

The 'Speed & Path Controller' block performs all functions described in Section3.2. This block's overview is shown illustrated in Figure B.9. This block has four subsystems. The first subsystem consists of the bang-bang controller used to provide translational control. This subsystem is illustrated in Figure B.10. The orientation controller, or quaternion controller is shown in Figure B.11 depicts how the Lyapunov equation derived in Section 3.2.4 is used to control the satellite quaternions. Both of these controllers are subject to system non-linearities as described in Section 3.2.6. Figure B.12 shows how these two controllers are modified these non-linearities. Figure B.15 shows an overview of how the two control signals are merged and converted into one thrust vector for the satellite. The signal to thrust conversion consists of two additional subsystems. These subsystems provide insight into the logic for converting translations and rotations into a series of zeros and ones to represent the thrust profile. Figure B.16 and Figure B.17 show a series of 'if/else' statements are used to convert the control force and control torque signals into a thrust profile.

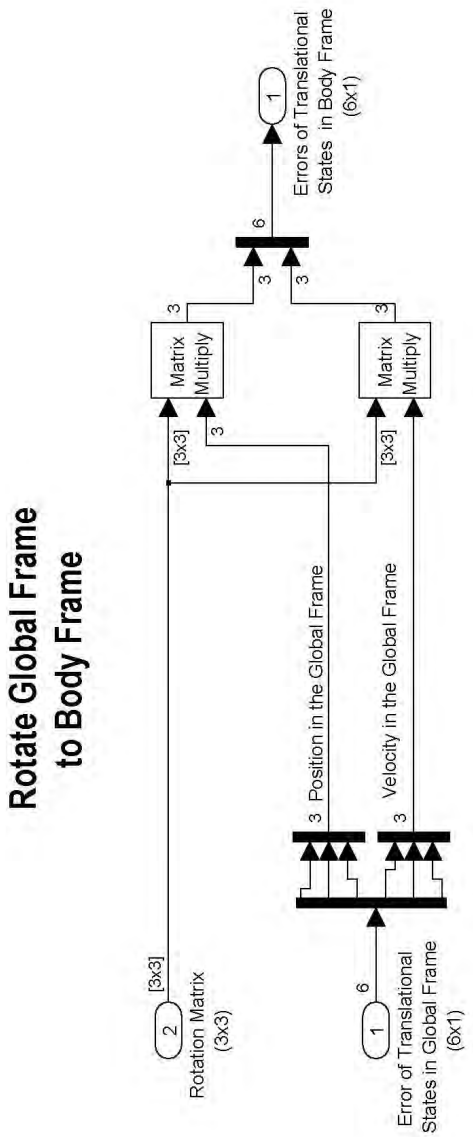

Figure B.6: Convert Translational Errors to the Body Frame from the Global Frame

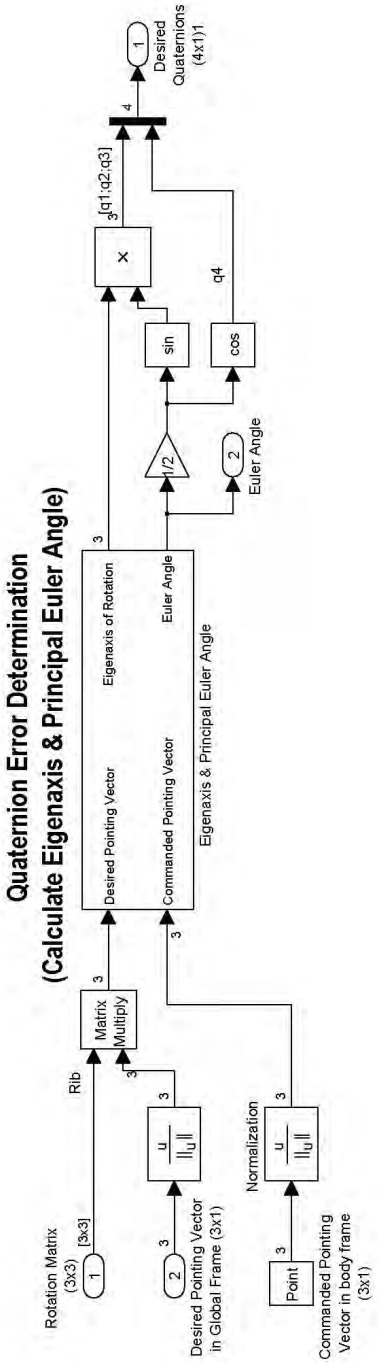

Figure B.7: Determine Quaternion Error

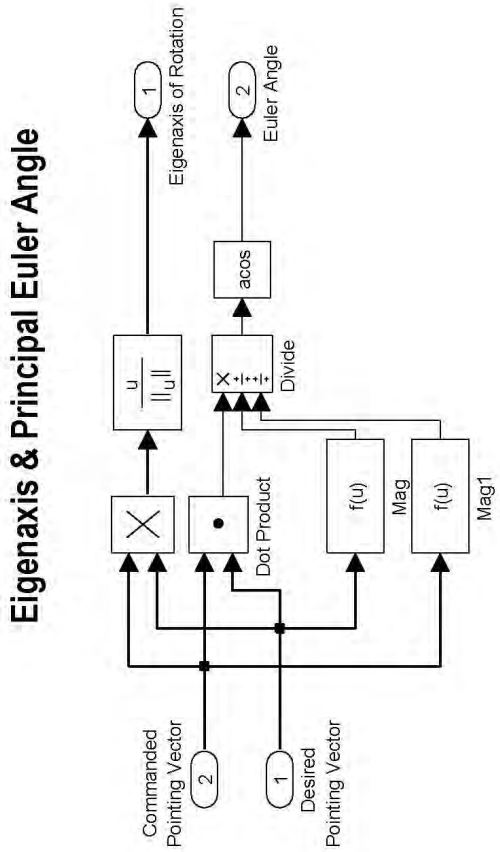

Figure B.8: Determine Eigenaxis of Rotation & Principal Euler Angle

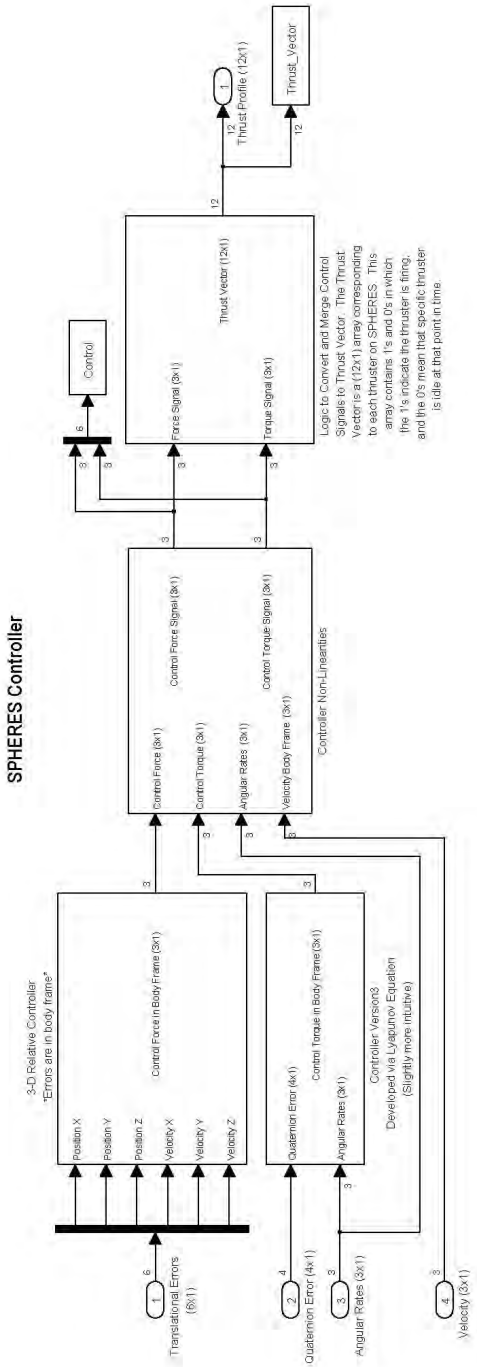

Figure B.9: Speed & Path Controller Overview

B.2.1 Control Algorithm. The translational states are controlled with the use of a bang-bang controller. This is shown in Figure B.10. In addition, Figure B.11 how the spacecraft quaternions are controlled using the controller derived from the Lyapunov function developed in Section 3.2.4.

B.2.2 Control Non-Linearities. Once the control signal has been created to correct both the translational and rotational errors, the signal is run through the necessary non-linearities. This is shown in Figure B.12. In addition, both control signals are passed through a rate limiter. This block checks to ensure the translational and rotational speeds of the satellite do not exceed constraints.

B.2.3 Control Signal Processing. Finally, before the control signals can interact with the SPHERES plant to make the required corrections, the signals must be converted into a thrust vector. The general process for this is described in Figure B.15. Since, specific thruster pairs cause the satellite to translate and different pairs cause the satellite to rotate, this process is broken into two portions. The translational component is shown in Figure B.16, and the rotational piece is included in Figure B.17.

B.3 SPHERES Plant

The 'SPHERES Plant' block performs all functions described in Section2.5. This block's overview is shown illustrated in Figure B.18. This block has two subsystems each with a number of additional subsystems. The first subsystem derives how the thrusters apply forces and torques on the satellite. The diagrams of this process are included in Figure B.19. The second subsystem uses the applied forces and torques to update the state vector of the satellite. This includes two parts. The satellites position and velocity must be updated as well as the satellites quaternions and angular rates.

B.3.1 Calculate Force & Torque from Thrust. Before the plant can update the state vector, it needs to determine how the thrusters are causing the satellite

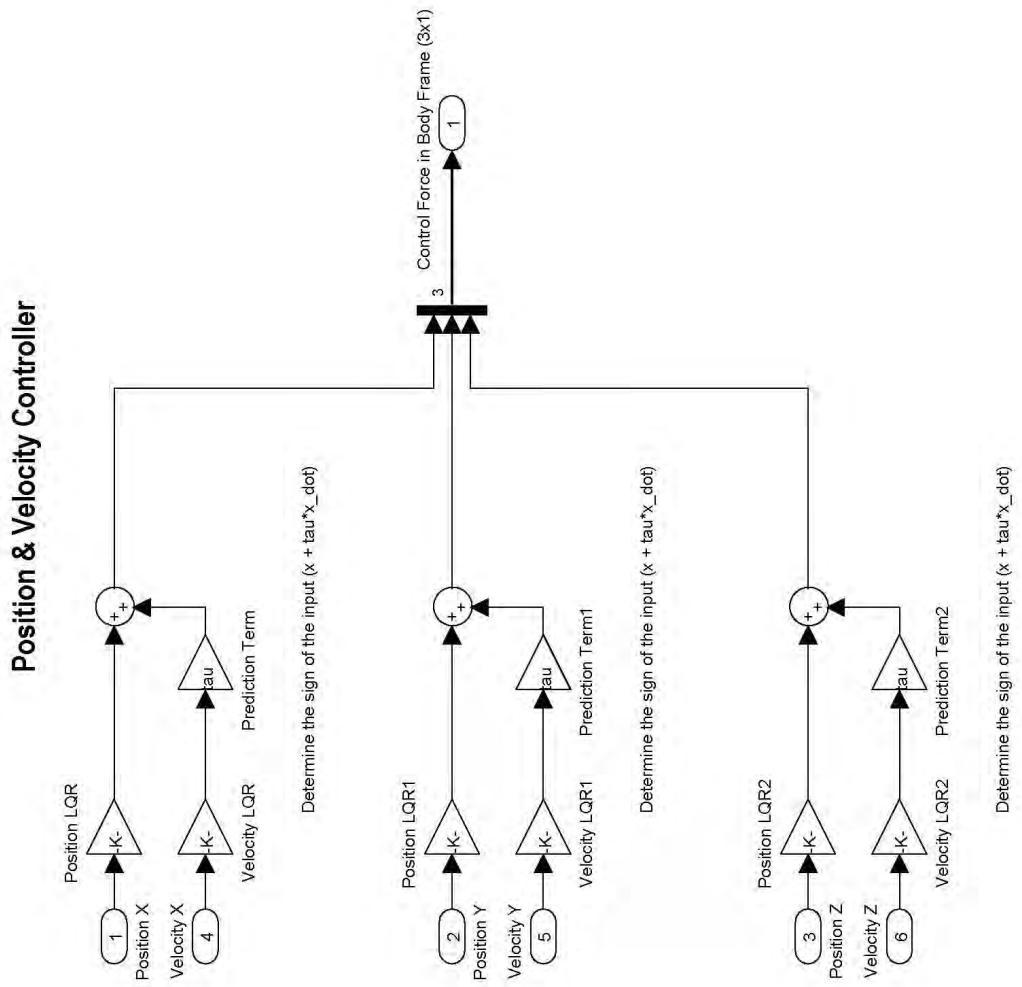

Figure B.10: Translational Controller

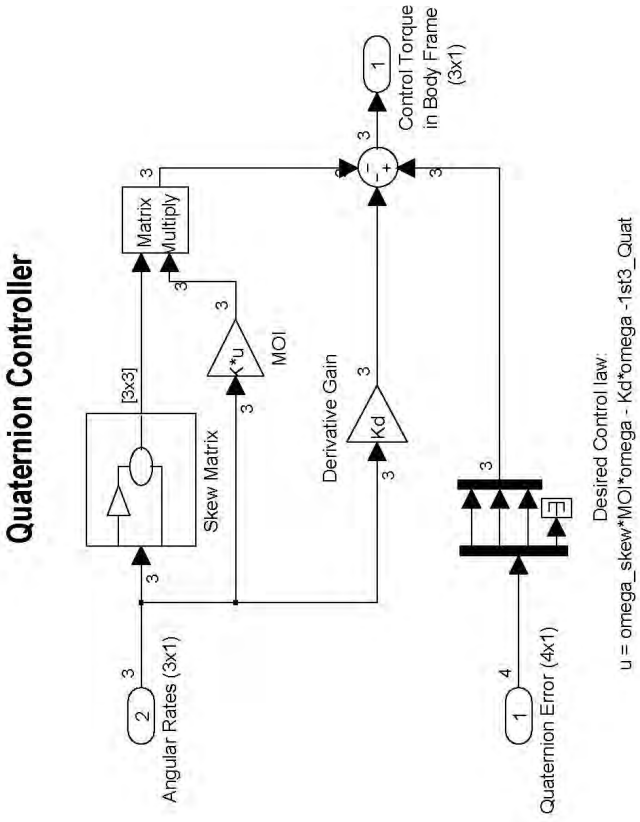

Figure B.11: Quaternion Controller

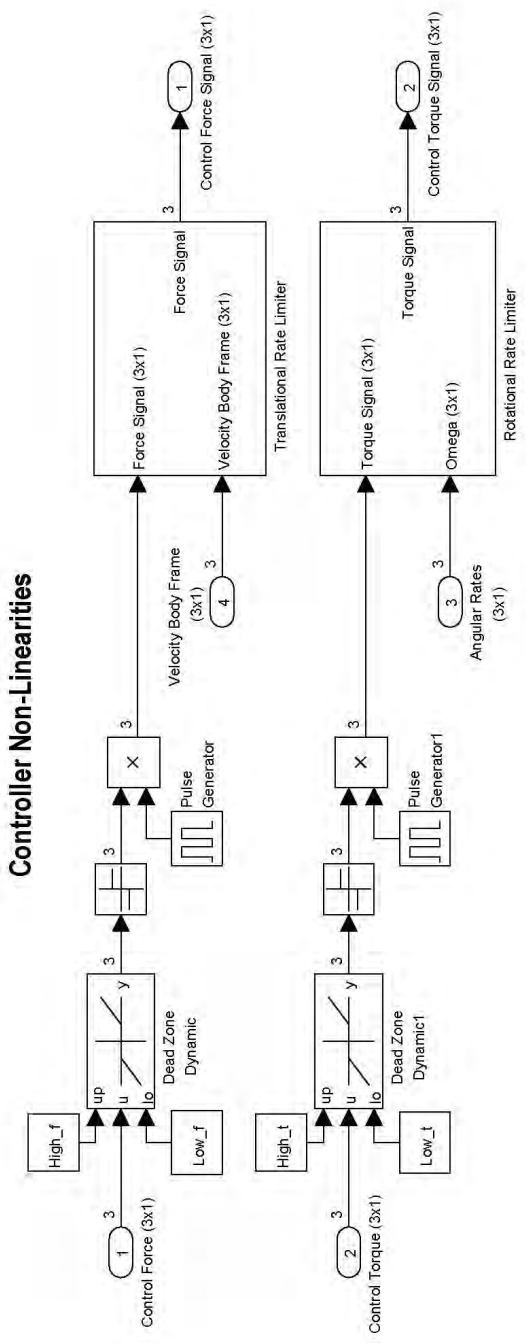

Figure B.12: Controller Non-Linearities

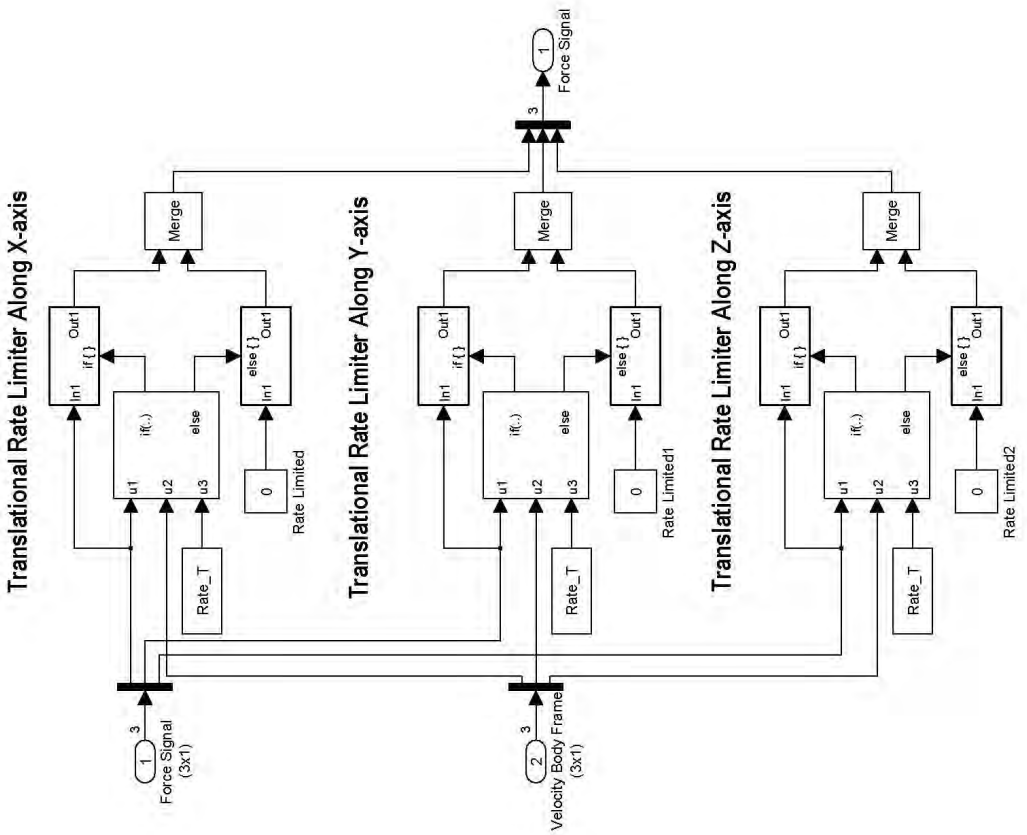

Figure B.13: Translational Rate-Limiter

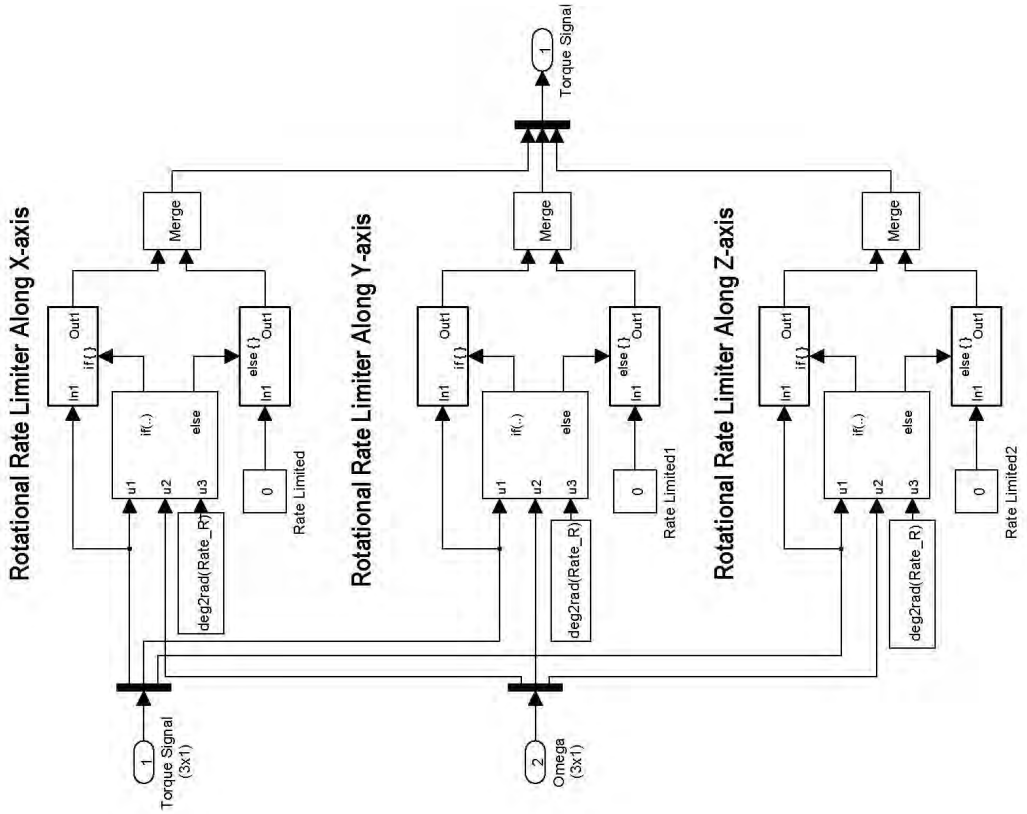

Figure B.14: Rotaional Rate-Limiter

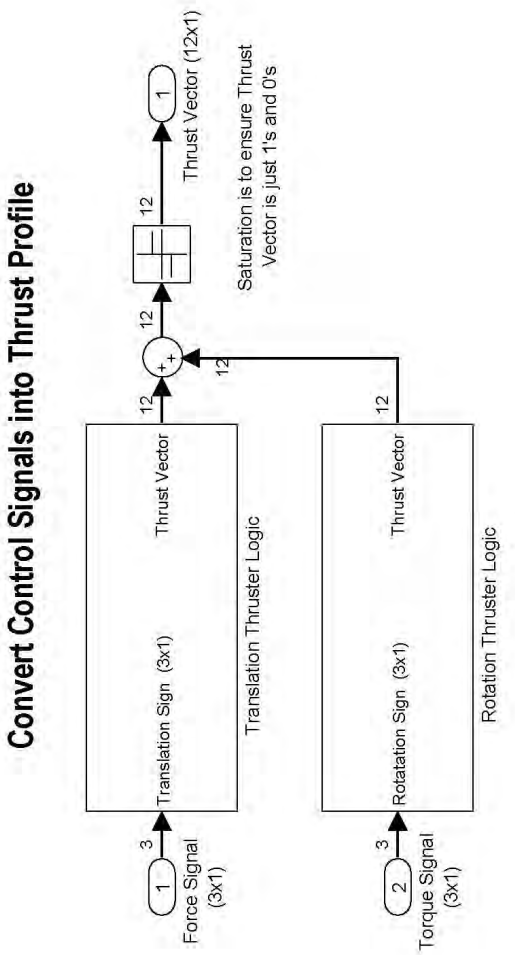

Figure B.15: Convert Control Signal to Thrust Vector

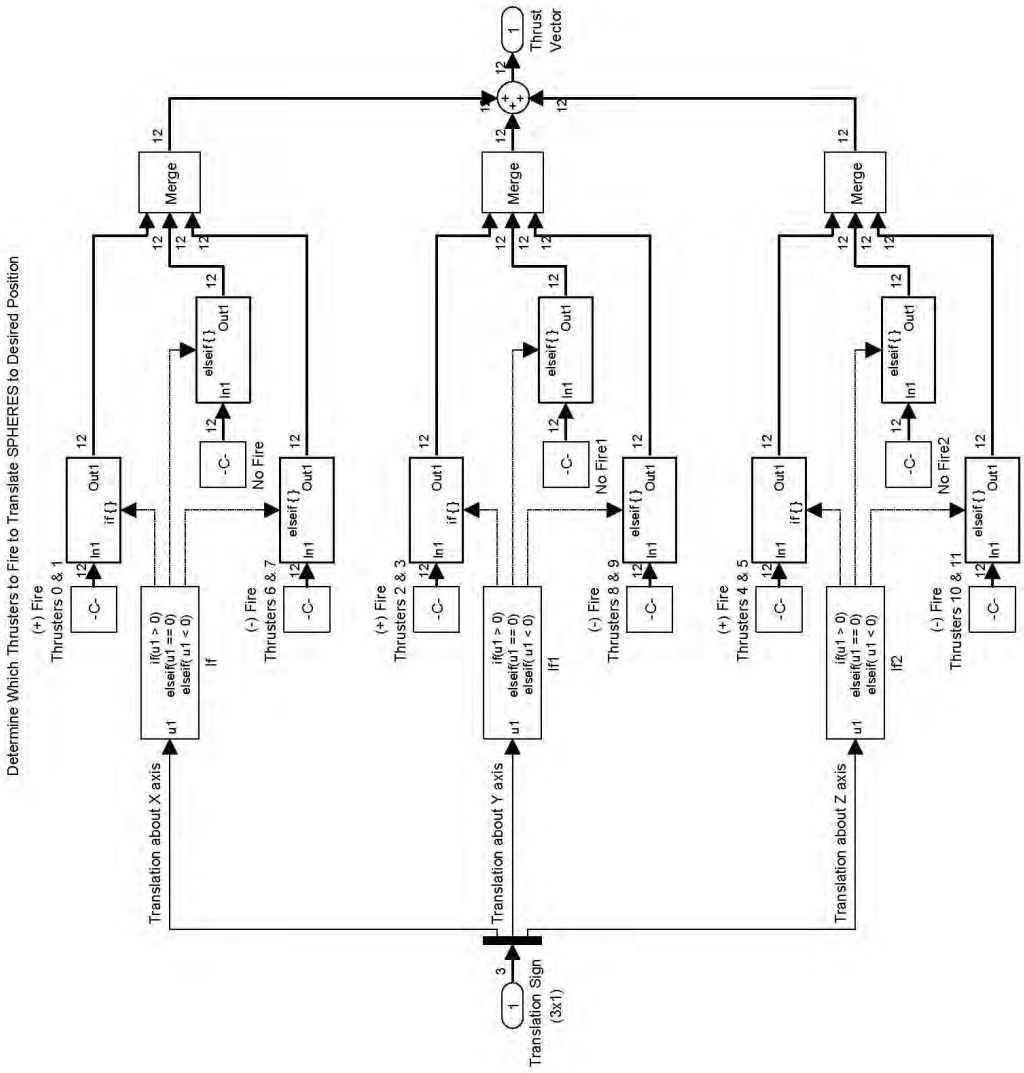

Figure B.16: Convert Control Force to Thrust Vector

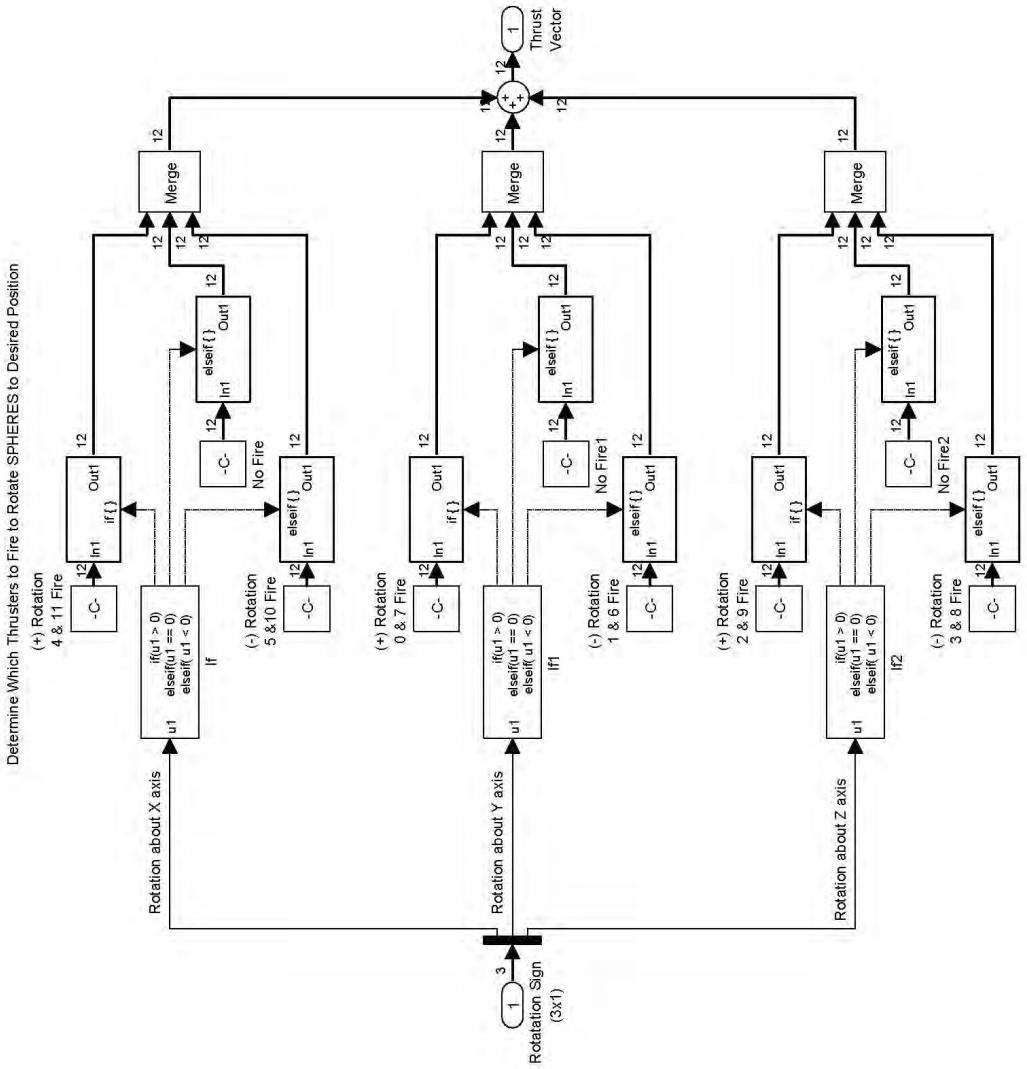

Figure B.17: Convert Control Torque to Thrust Vector

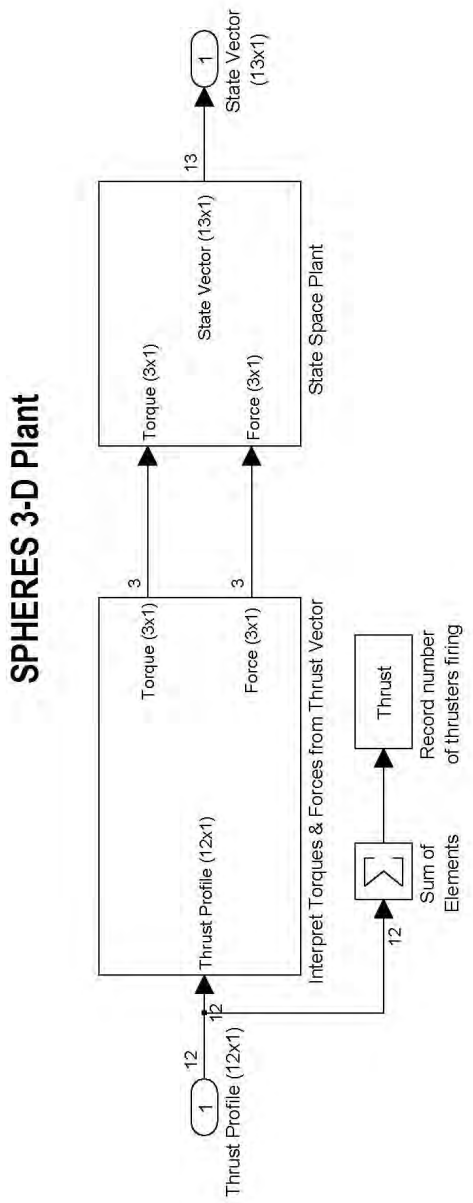

Figure B.18: Diagram of SPHERES Plant

to move. The force and torques generated from the thrusters are determined in Figure B.19.

B.3.1.1 Convert Thrust to Force. The force vector is recreated using a number of 'if/else' statements. These statements are applied determine if a force is applied to each axis in the body frame. Figure B.20 provides an overview of this process and Figures B.21, B.22, and B.23 provide further detail for each body frame axis.

B.3.1.2 Convert Thrust to Torque. The torque vector is recreated using a number of 'if/else' statements. These statements are applied determine if a torque is applied about each axis in the body frame. Figure B.24 provides an overview of this process and Figures B.25, B.26, and B.27 provide further detail for each body frame axis.

B.3.2 Update State Vector. Once the force and torque vectors have been determined from the thrust profile, the satellite state vectors can be updated. in a general sense, this is accomplished in Figure B.28.

B.3.2.1 Update Position & Velocity. The position and velocity of the satellite are updated in Figure B.29. It is worth noting that the actual update is performed in the global frame, but at the moment the force vector is in the body frame. Figure B.30 shows how the force vector is rotated into the global frame before the position and velocity are updated using state-space techniques.

B.3.2.2 Update Quaternions & Angular Rates. Both the quaternions and the angular rates are updated in Figure B.32. This process uses the equations explained in Section 2.5.2. Since a number of subsystems are embedded within Figure B.32, the quaternion subsystem will be discussed first and then the angular rates are explained second.

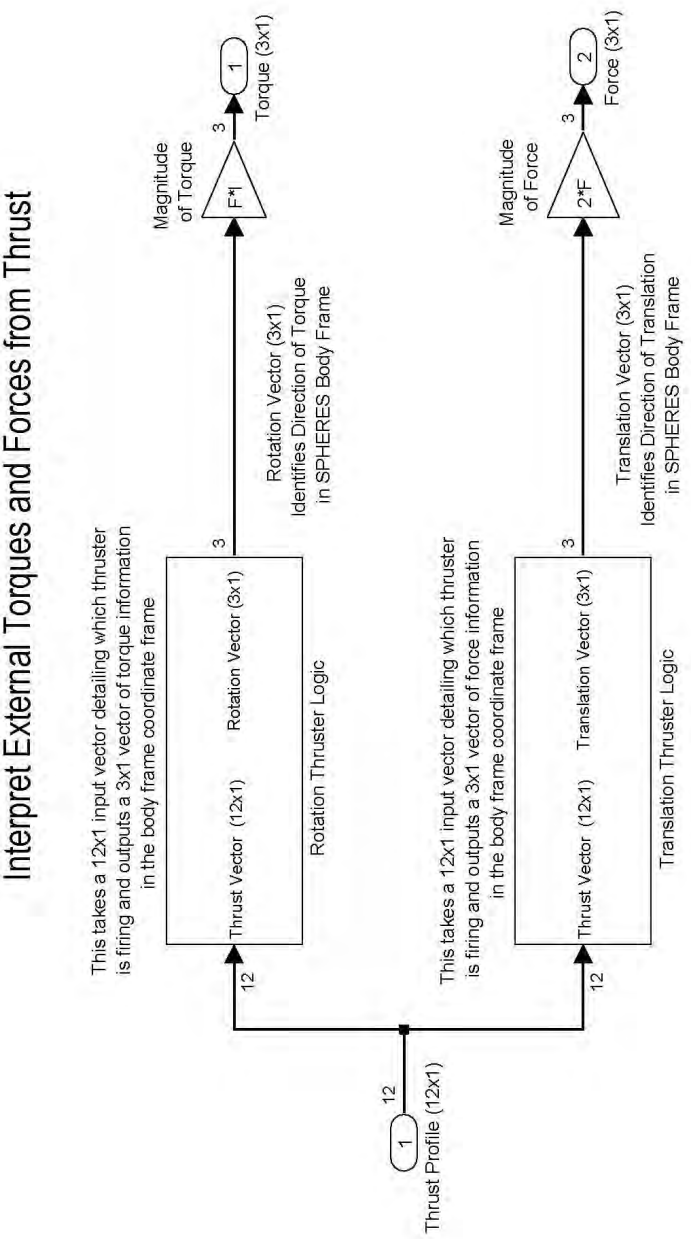

Figure B.19: Convert Thrust Vector into Applied Force & Torque

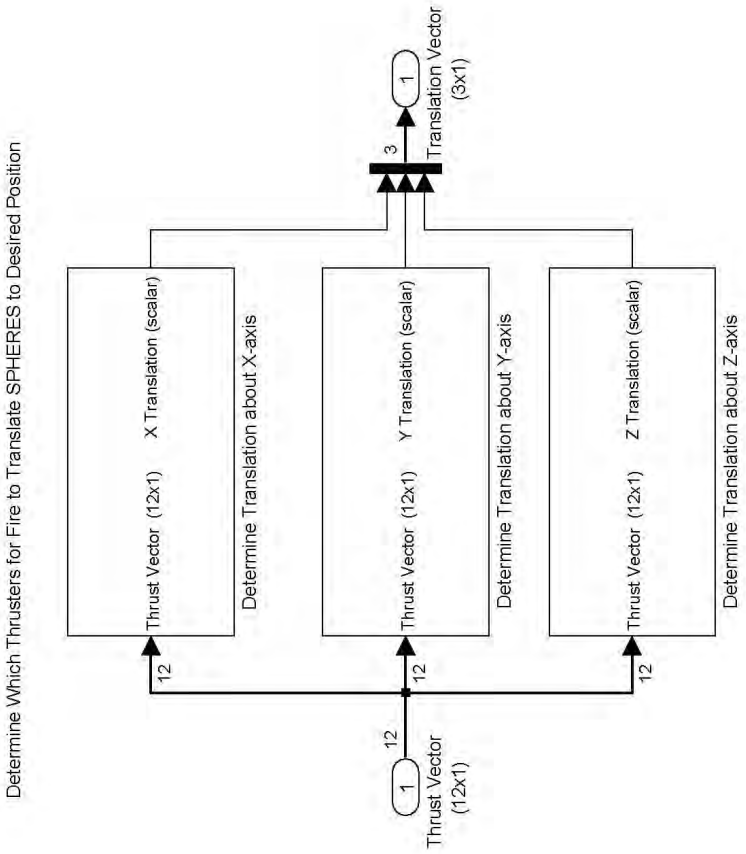

Figure B.20: Convert Thrust Vector into Applied Force

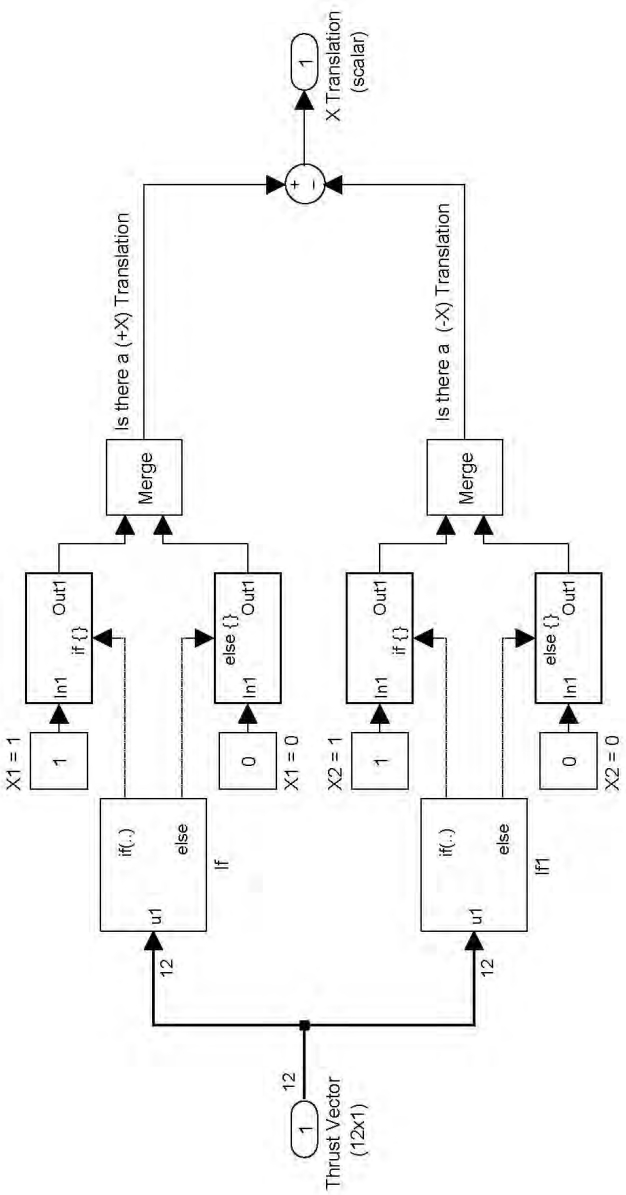

Figure B.21: Calculate Applied Force Along X-axis

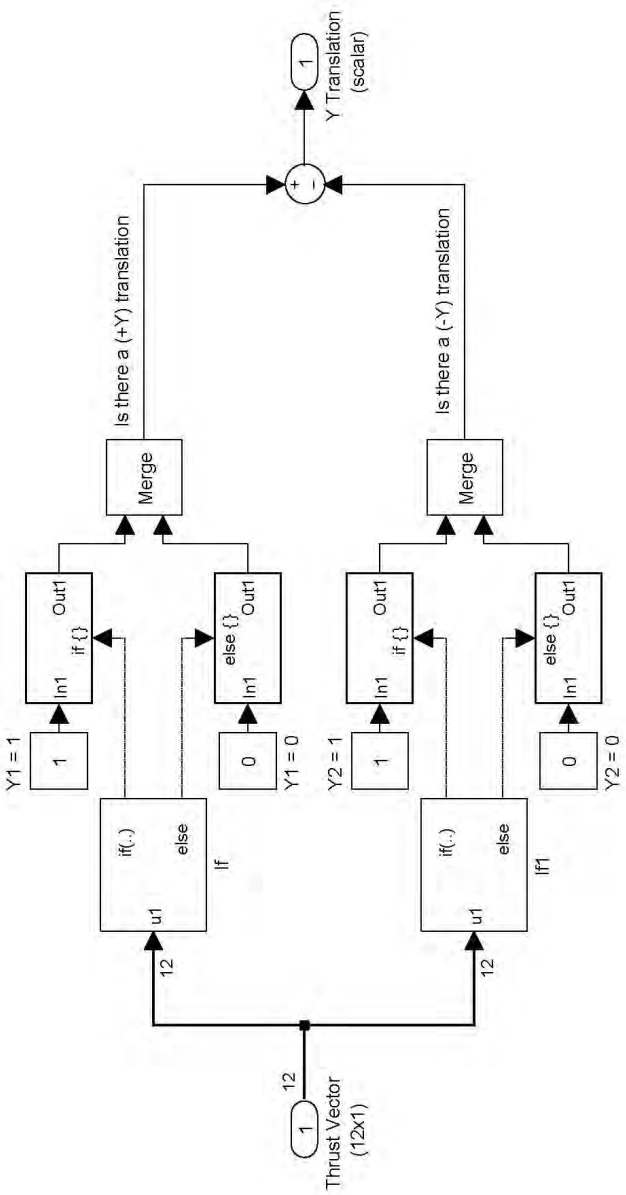

Figure B.22: Calculate Applied Force Along Y-axis

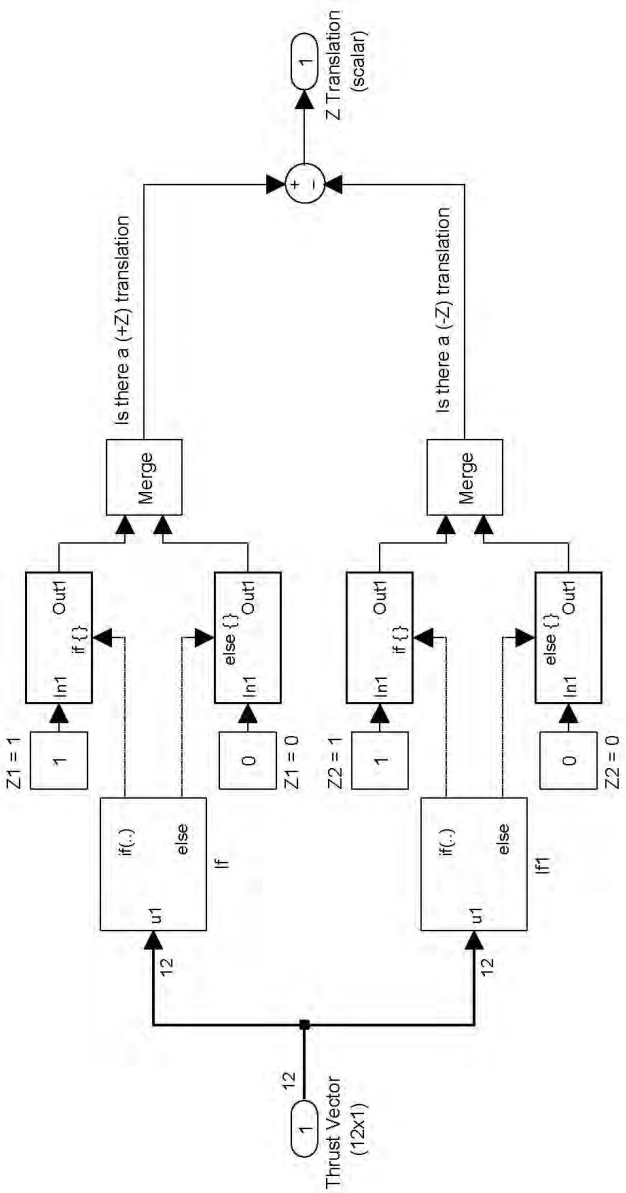

Figure B.23: Calculate Applied Force Along Z-axis

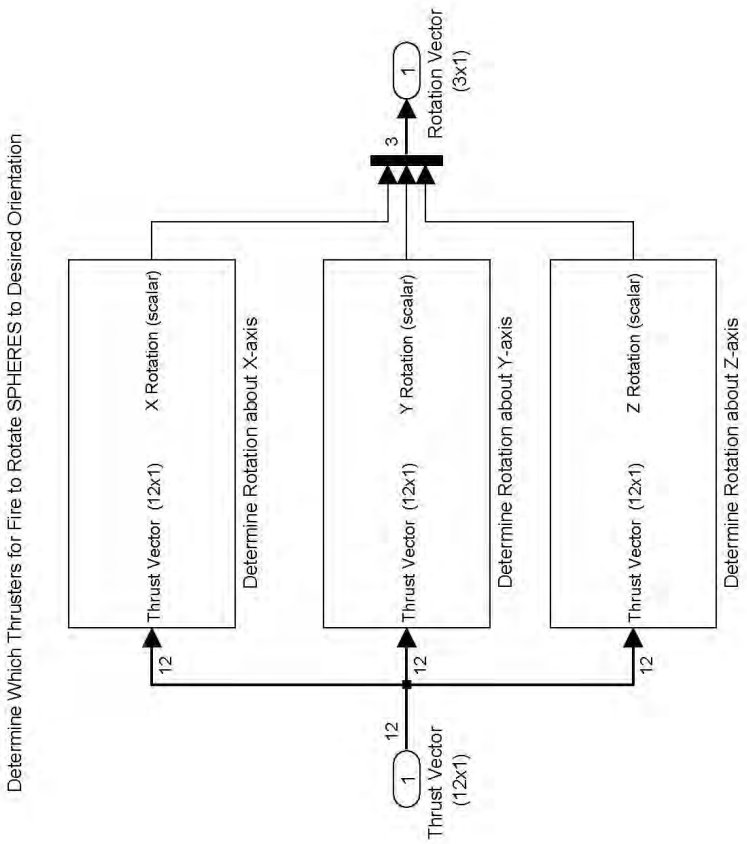

Figure B.24: Convert Thrust Vector into Applied Torque

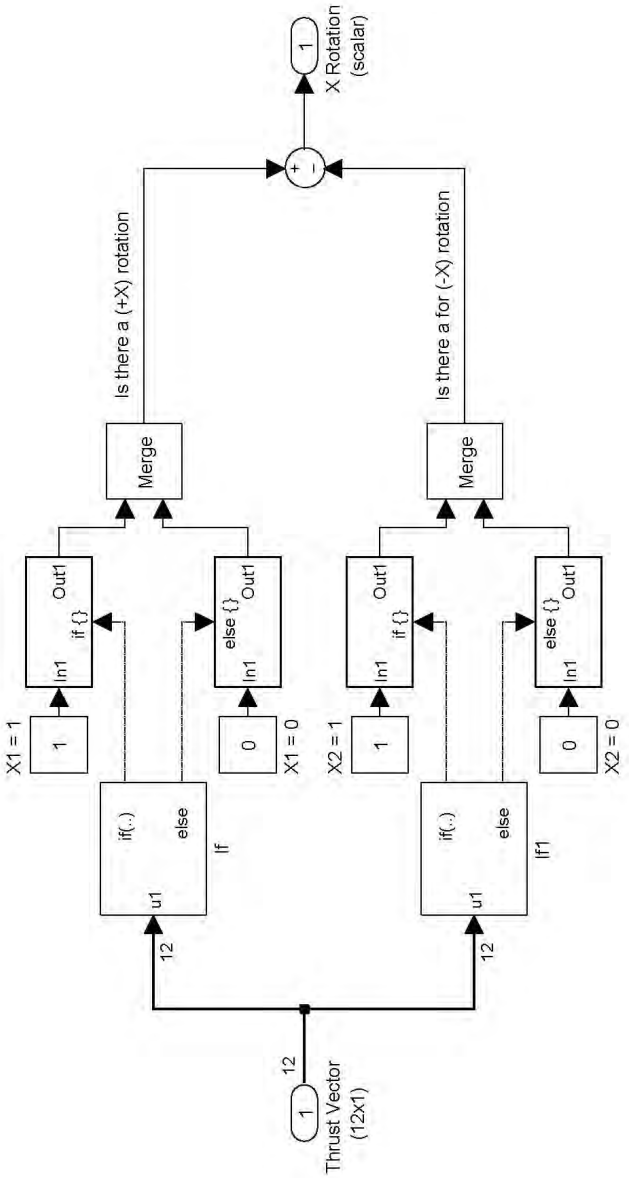

Figure B.25: Calculate Applied Torque Along X-axis

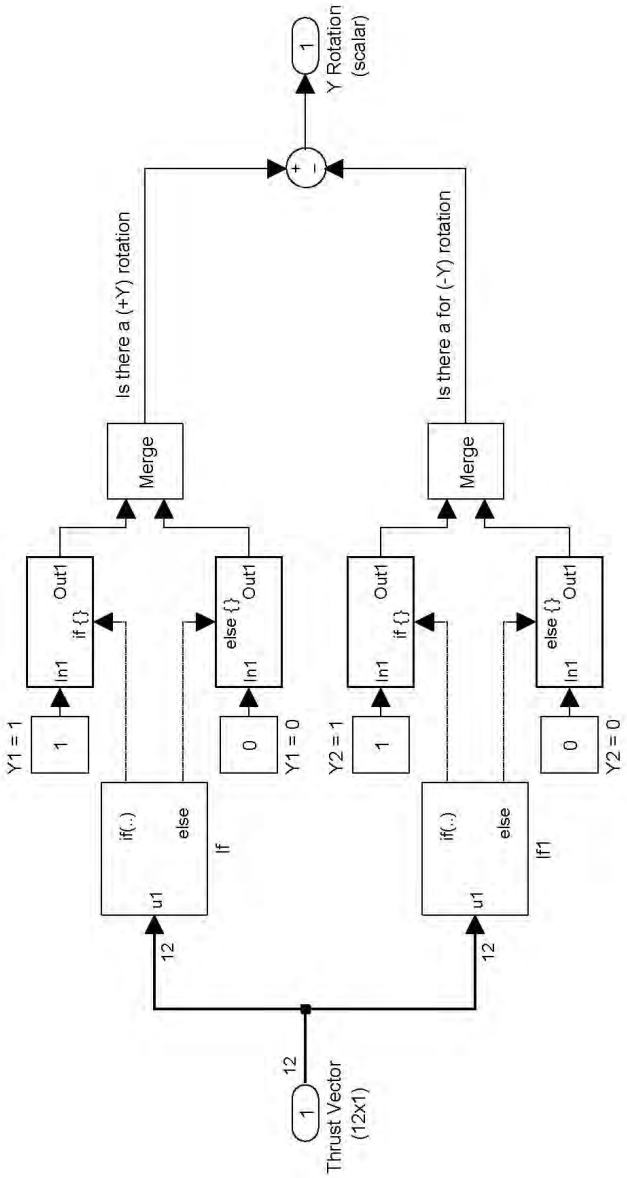

Figure B.26: Calculate Applied Torque Along Y-axis

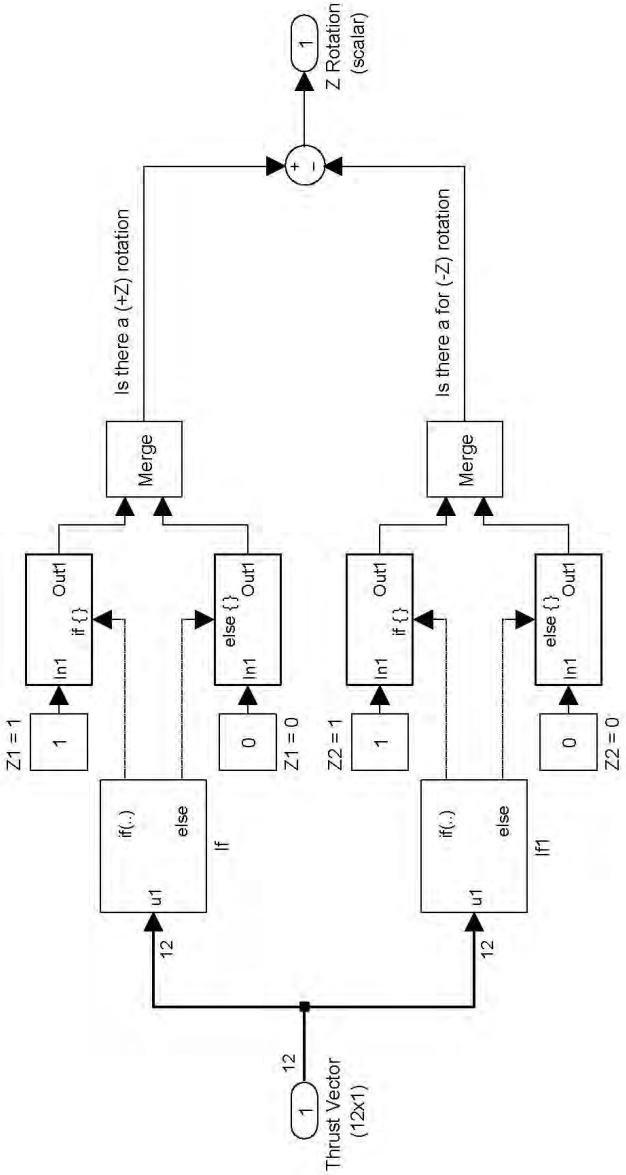

Figure B.27: Calculate Applied Torque Along Z-axis

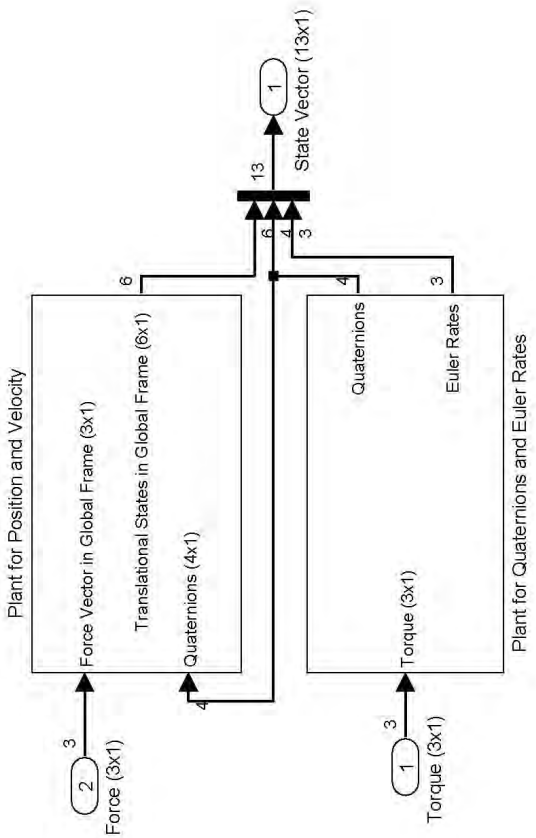

Figure B.28: Update State Vector via Rate Equations

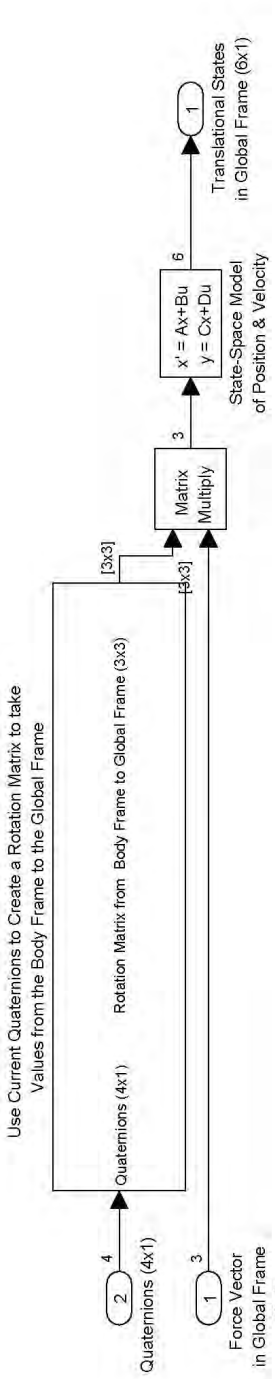

Figure B.29: Convert Translation States to Global Frame & Apply to State Space Model

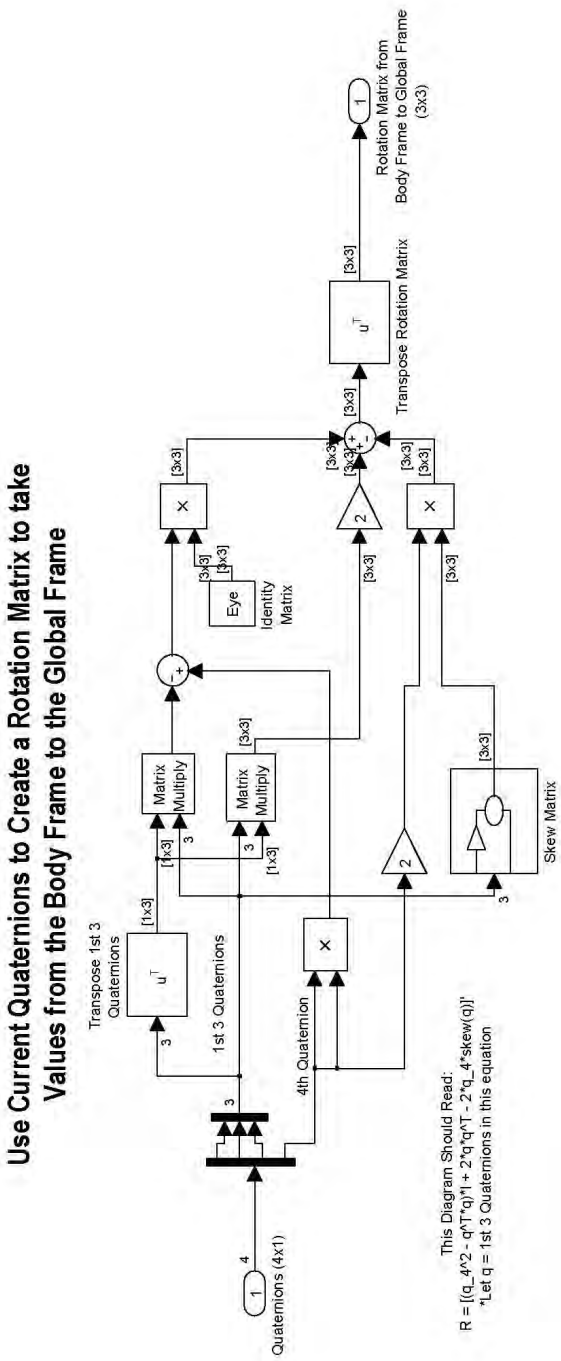

Figure B.30: Create Rotation Matrix to Go From Body Frame to Global Frame

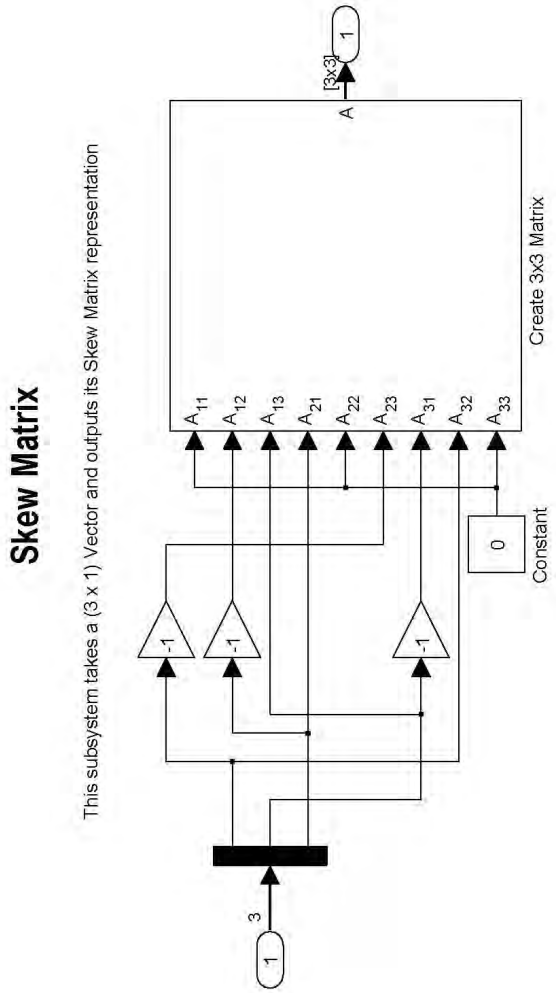

Figure B.31: Skew Matrix

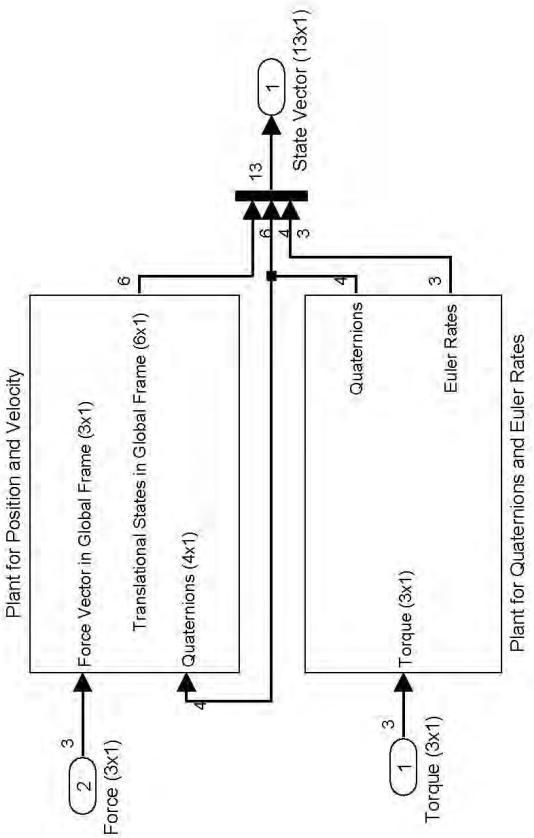

Figure B.32: Quaternions & Euler Rates of Plant Model

The quaternions are updated in two parts. The first three quaternions are updated together and the fourth quaternion is updated using a separate equation. Figure B.33 provides the overview of this process. Additionally, Figure B.34 describes how the first three quaternions are updated and Figure B.35 shows how the fourth quaternion is updated.

In addition to updating the quaternion rates, the angular rates are updated as well. Figure B.36 shows how the rate of the angular rates is integrated to determine the new angular rates, and Figure B.37 describes the process for updating the rate of the angular rates. This process uses Equation 2.50.

B.4 Inputs & Outputs

The last two subsystems to be mentioned from Figure B.1 are the 'User Inputs' and 'Outputs' blocks. The 'User Inputs' block loads the users inputs from a table and inserts them into the simulation as shown in Figurefig:User Inputs. In addition, the 'Outputs' block saves the satellite state vector in both the global and body reference frames as depicted in Figure B.39.

B.4.1 Breakdown of Output Subsystems. Although Figure B.39 shows the general layout of how the satellite state vector is saved, three of subsystems internal to this process are also detailed to fully illustrate how the 'Outputs' block of Figure B.1 operates.

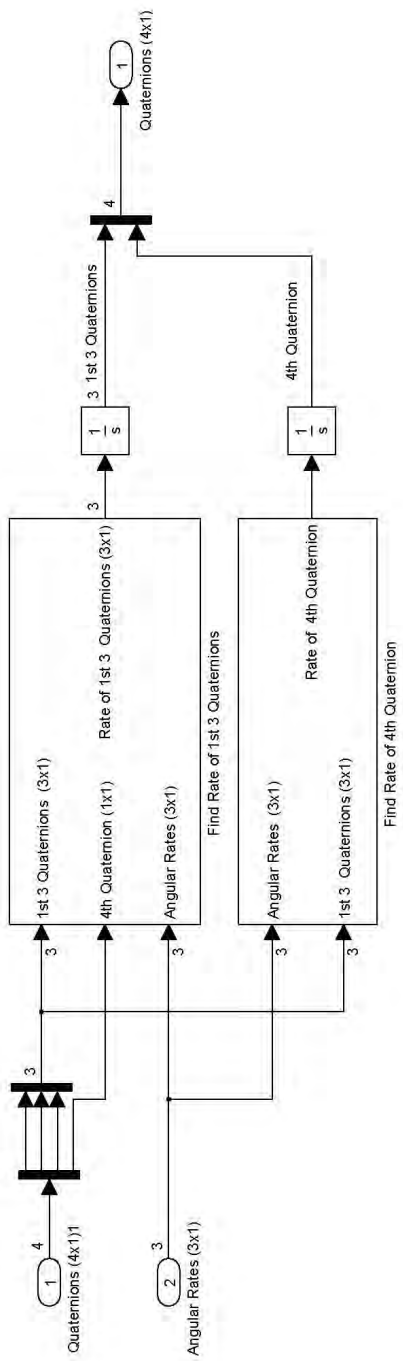

Figure B.33: Overview of Quaternion Update

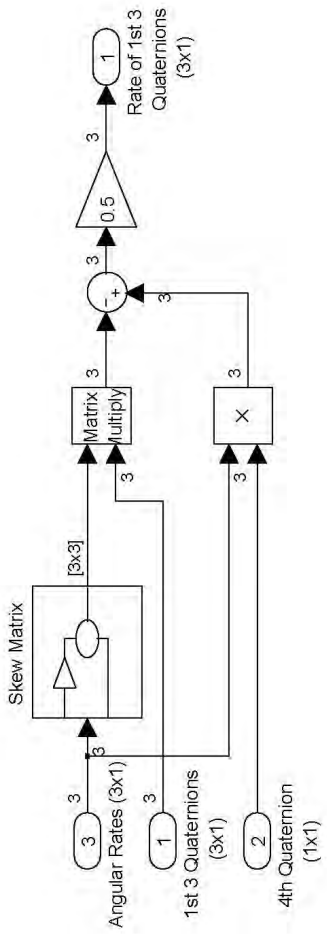

Figure B.34: Update 1^{st} Three Quaternions

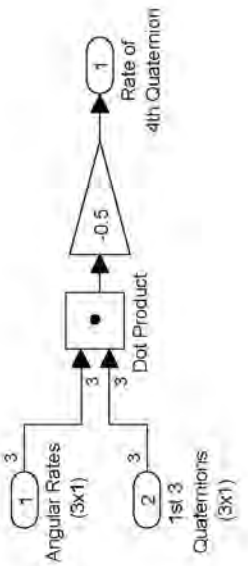

Figure B.35: Update 4^{th} Quaternion

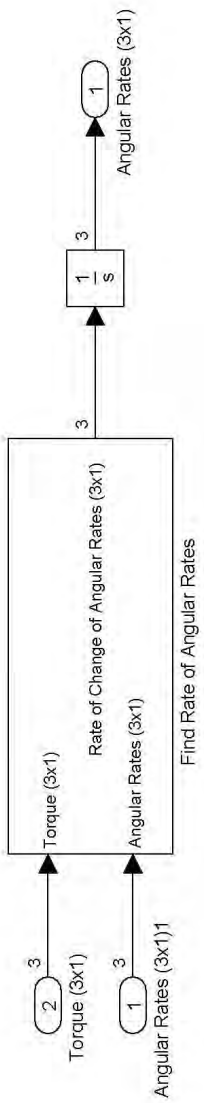

Figure B.36: Overview of Angular Rate Update

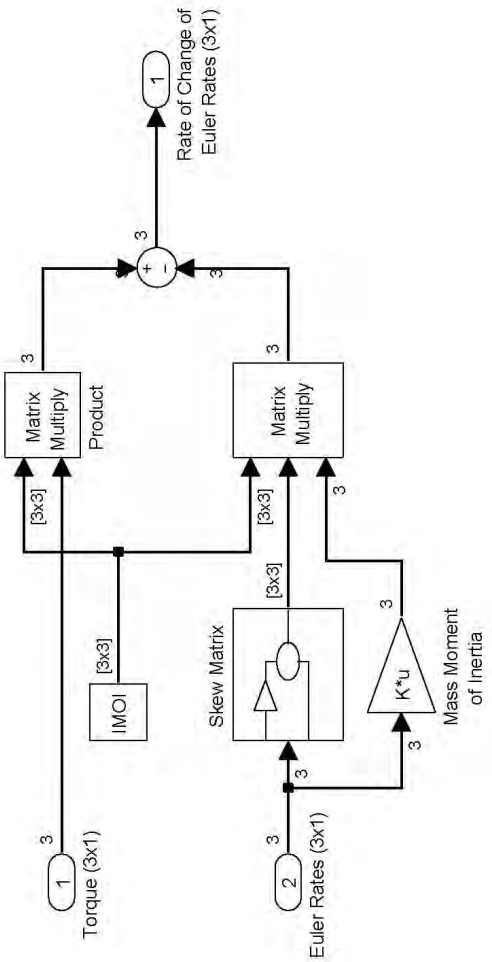

Figure B.37: Update Rate of Angular Rates

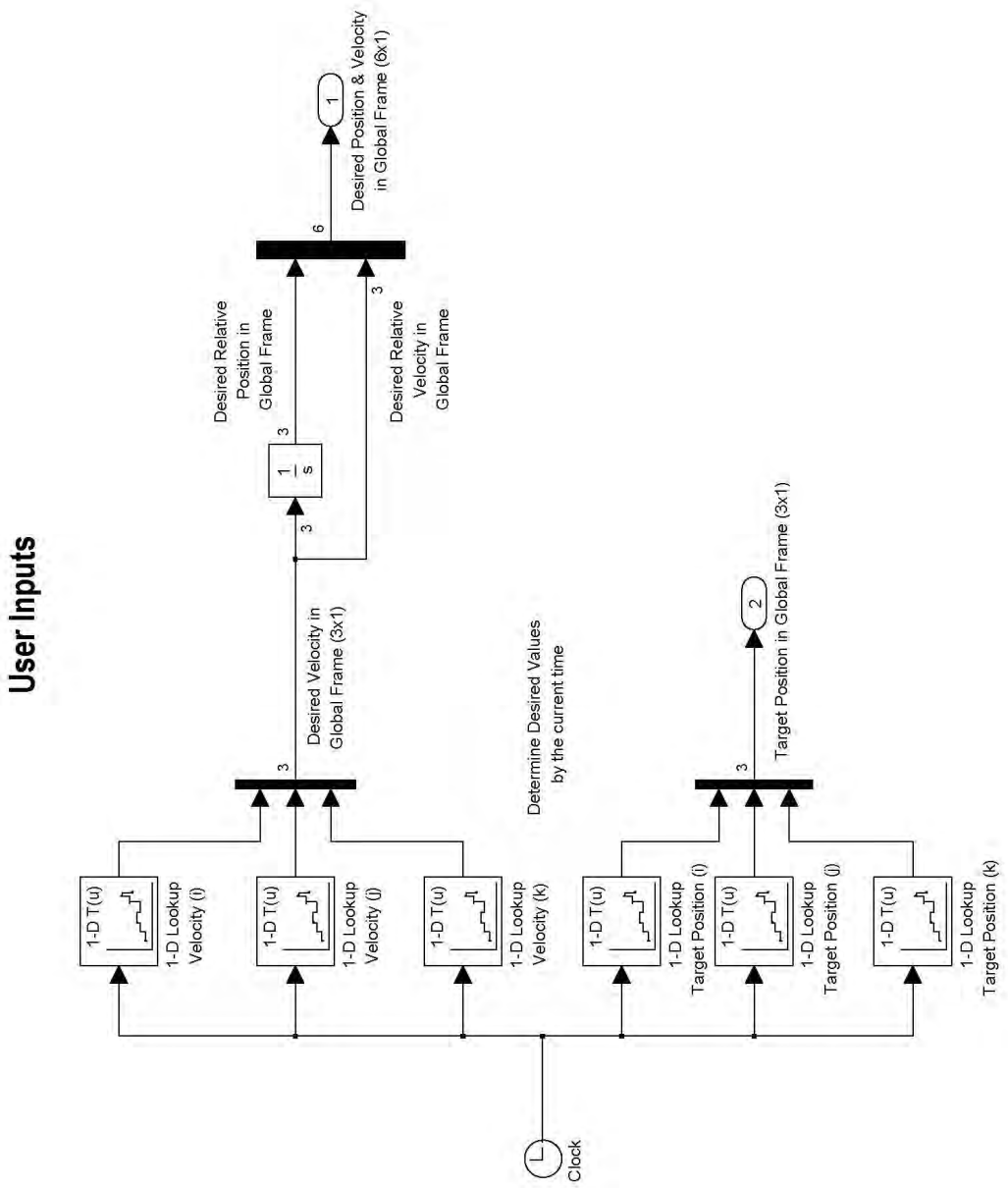

Figure B.38: User Inputs

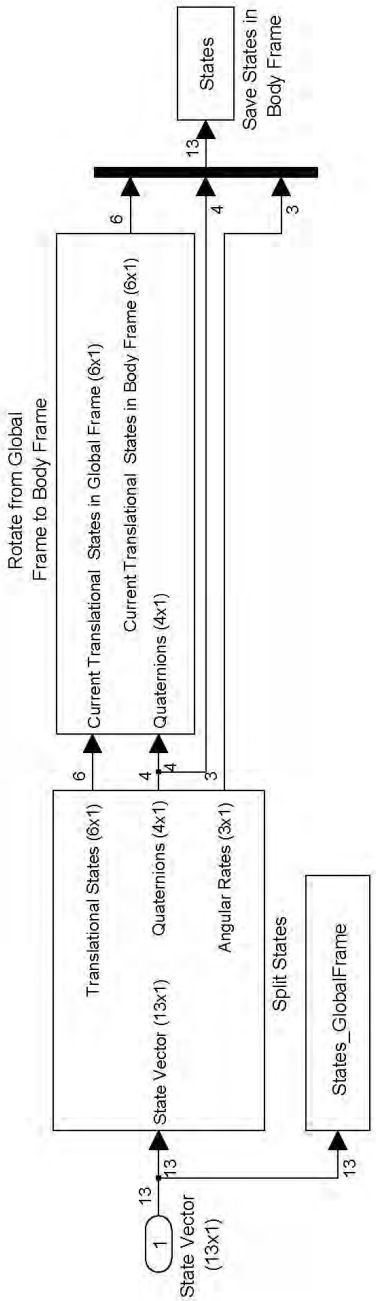

Figure B.39: Outputs Overview

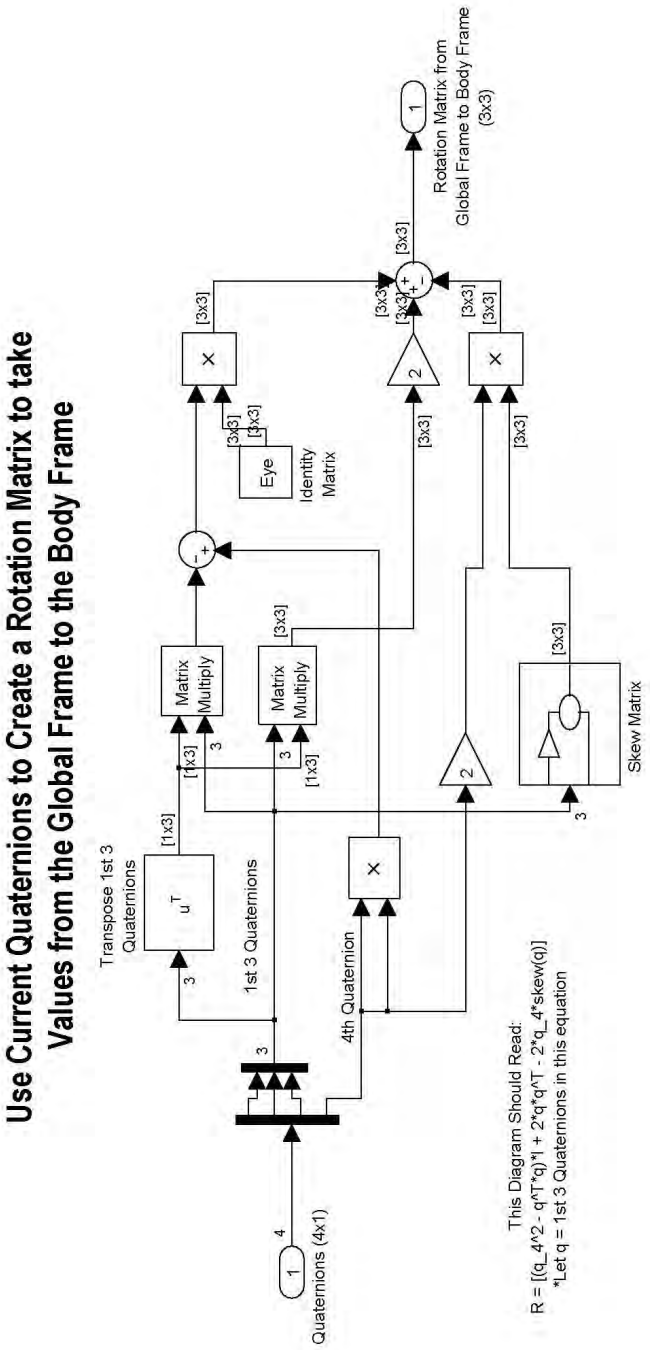

Figure B.40: Rotate Translational State Information to Satellite Body Frame

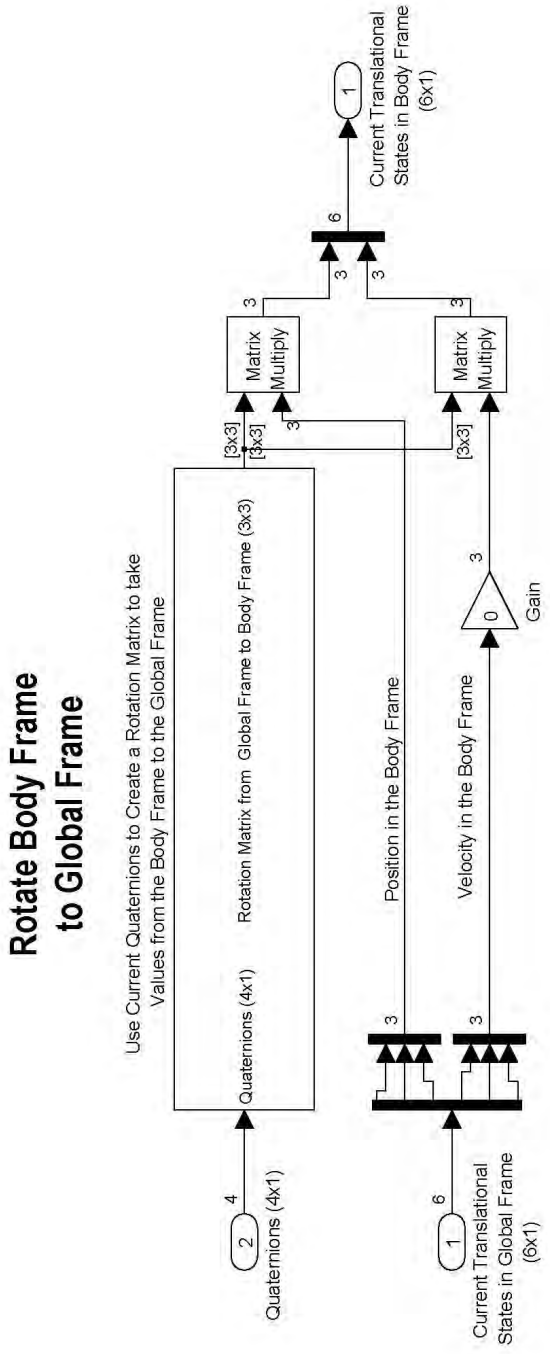

Figure B.41: Rotation Matrix to Rotate Information from Global Frame to Body Frame

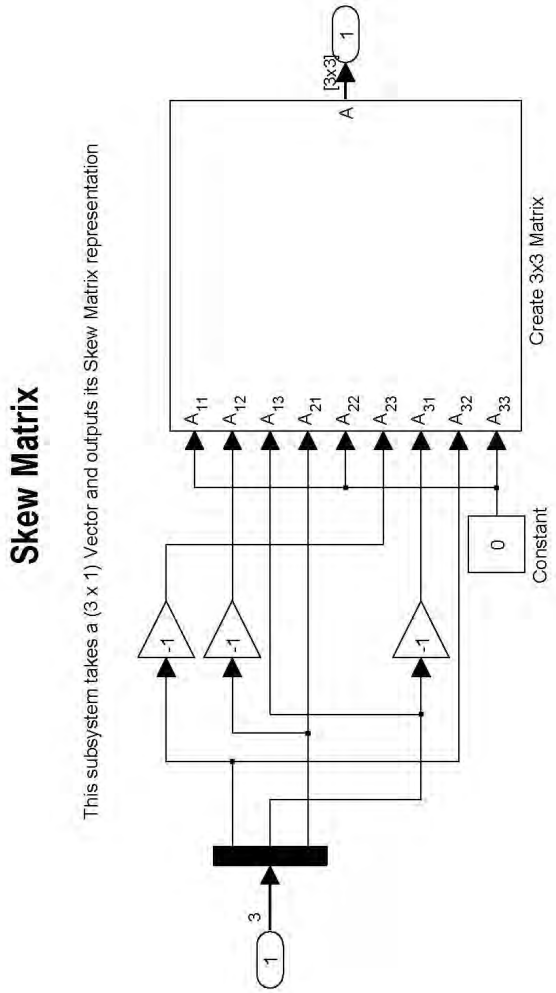

Figure B.42: Skew Matrix

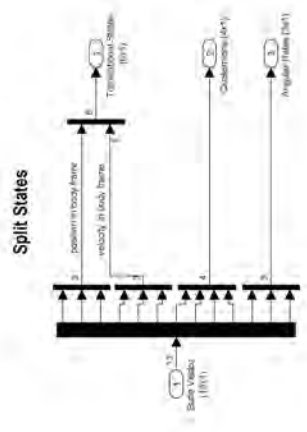

Figure B.43: Split State Vector

This appendix serves to include the MATLAB® scripts that were used to optimize the gain values used throughout the control algorithm. These scripts are included to provide one with an in-depth understanding of how this controller was developed, and to serve as a launching point for future work. Specifically this code could be used to determine optimal values for this controller under different circumstances.

C.1 Script to Optimize LQR Weights

The following code is titled 'SPHERES_LQR_TradeStudy.m' and is included below for reference within this thesis. This code was used to determine the desired weights for the LQR. Anyone wishing to modify the weights for different criterion should consider working from this code.

Listing C.1: Appendix3/SPHERES_LQR_TradeStudy.m

```
%%%%%%%%%%%%%%% SPHERES LQR Weighting Trade Study %%%%%%%%%%%%%%%%%%
%
% Trade Study for 1-D Model of SPHERES Bang-Bang Controller
%
% Author: Sam Barbaro        ENY-3                    26 Oct 2011
%
% Purpose: This script feeds constants & variables into the
% simulation SPHERES_1D_Model_Relative_LQR.mdl for various
% weighting ratios of Q & R and then records how the percent
% overshoot and settling time of the relative position & velocity
% errors change.
%
% Model: This simulation considers the thrusters numbered 0, 1, 6,
% & 7 as they are the ones that when coupled translate SPHERES in
% the X-axis and rotate SPHERES about its Y-axis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
clc; clear; close all

%% User Variables (CHECK BEFORE EACH RUN)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% USER MAKES CHANGES HERE %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set Initial Conditions
Pos_0     = 0;          % initial relative position of SPHERES [m]
Vel_0     = 0;          % initial relative velocity of SPHERES [m/s]
Initial   = [Pos_0; Vel_0]; % Simulation input for State Space

% Determine Basic Velocity Path
Vel_Des   = .2;         % desired relative velocity [m/s]
Vel_Time  = 1;          % time at which velocity is commanded [sec]

% Weighting Conditions
PosWght   = 5;
VelWght   = .001:.001:1;
FuelWght  = .3;

% Find length of the weighting condition that is changing
dim       = length(VelWght);

% Set constant weighting conditions to same length as the
% variable weighting condition
Q1        = PosWght.*ones(1,dim);
Q2        = VelWght.*ones(1,dim);
R1        = FuelWght.*ones(1,dim);

% Label which weighting condition is changing
% var       = 'Position Weight';
var       = 'Velocity Weight';
% var       = 'Control Weight';
```

```
% rat       = 'Position/Velocity';
% rat       = 'Velocity/Position';
rat       = 'Velocity/Control';

idw       = Q2./R1;     % identify how weight ratio changes

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% END TYPICAL USER CHANGES %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Constants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mass Moments of Inertia
Eye           = [1 0 0; 0 1 0; 0 0 1];   % Identity Matrix
MOI_Wet = [ 2.30e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
             9.90e-5   2.42e-2  -2.54e-5;% SPHERES MOI w/ full tank
            -2.95e-4  -2.54e-5   2.14e-2;];
MOI_Dry = [ 2.19e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
             9.90e-5   2.31e-2  -2.54e-5;% SPHERES MOI w/empty tank
            -2.95e-4  -2.54e-5   2.13e-2;];
InMOI_Wet     = inv(MOI_Wet);            % shorthand
InMOI_Dry     = inv(MOI_Dry);            % shorthand
OneD_MOI_Wet  = MOI_Wet(2,2);            % MOI used for 1-D example
OneD_MOI_Dry  = MOI_Dry(2,2);            % MOI (dry) used for 1-D

% Mass of SPHERES
Mass_Wet   = 4.16;       % SPHERES mass with fuel in [Kg]
Mass_Dry   = 3.55;       % SPHERES mass without fuel in [Kg]
% Thruster Information
F    = 0.11;             % force of individual thruster [N]
l    = 0.193;            % length between thrusters [m]
% Define Dead Zone Range
                         % these values ensure translation stays
                         % w/in 1 cm of the desired value
High          = 0.002;  % values above this number aren't reset
```

```
Low          = 0.002;  % values below this number aren't reset
% SPHERES Plant
A            = [0 1; 0 0];
Bu           = [0;1/Mass_Wet];
C            = eye(2); % assumes all states provided by estimator
D            = zeros(2,1);% control doesn't directly affect output

%% Develop LQR & Simulate

%%%%% Test Run to Help Debug... assumes only one case%%%%%%%%%%%%
% Q = [5 0; 0 0.09]; R = 0.3;
% [K,S,E]    = lqr(A,Bu,Q,R);    % linear quadratic controller
% sim('SPHERES_1D_Model_Relative_LQR1');
% %%
% plot(simout1.time(:,1),simout1.signals.values(:,1),'r',...
%      simout1.time(:,1),simout1.signals.values(:,2),'b');
% %title('Error vs. Time for 1-D SPHERES Model with LQR Control');
% xlabel('Time [sec]'); ylabel('[m] or [m/sec]');
% legend('Position','Velocity','Location','NorthEast');
% set(gca,'fontsize',19)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PO    = zeros(dim,2);                %
Ts    = zeros(dim,2);                % pre-set variables for speed
Tbrn  = zeros(dim,1);                %

% Test Each Case of Weights Considered for LQR Controller
for ctr = 1:dim;
    Q          = [Q1(ctr) 0; 0 Q2(ctr)];
                                     % build Q matrix
    R          = R1(ctr);            % build R matrix
    [K,~,~]    = lqr(A,Bu,Q,R);      % linear quadratic controller
    sim('SPHERES_1D_Model_Relative_LQR1');
                                     % run SPHERES simulation
```

```
        % Find the Percent Overshoot
        PO(ctr,1)   = 100*max(simout1.signals.values(:,1));
126     PO(ctr,2)   = 100*max(-simout1.signals.values(:,2));
        % Find the Settling Time (2% Criteria)
        if PO(ctr,1) < 2;
            Ts(ctr,1) = 0;
        else %PO(ctr,1) ≥ 2;
131         index1    =find(simout1.signals.values(:,1)≥.02,1,'last');
            Ts(ctr,1) =simout1.time(index1,1);
        end
        if PO(ctr,2) < 2;
            index2    =find(simout1.signals.values(:,1)≥.02,1,'last');
136         if isempty(index2) == 1
                Ts(ctr,2) = 0;
            else
                Ts(ctr,2) = simout1.time(index2,1);
            end
141     else %PO(ctr,2) ≥ 2;
            index2  =find(simout1.signals.values(:,2)≤-.02,1,'last');
            Ts(ctr,2)  = simout1.time(index2,1);
        end
        % Find the Amount of Time the Thrusters are Firing
146     Tbrn(ctr)  = sum(abs(simout1.signals.values(:,3)))*...
                     (simout1.time(2,1)-simout1.time(1,1));
    end

    %% Format Results
151
    % Plotting commands have been removed
```

C.2 Script to Optimize τ

The following code is titled 'SPHERES_Tau_TradeStudy.m' and is included below for reference within this thesis. This code was used to determine the desired

look-ahead gain for the bang-bang controller. Anyone wishing to modify this gain for different criterion should consider working from this code.

Listing C.2: Appendix3/SPHERES_Tau_TradeStudy.m

```
%%%%% SPHERES Translation Controller 1-D Phase Plane Analysis%%%%%
%
% 1-D Model for SPHERES Bang-Bang & LQR Controller
%
% Author: Sam Barbaro         ENY-3                    25 Oct 2011
%
% Purpose: This script feeds variables into the
% 'SPHERES_1D_Model_Relative_LQR_v2' simulation and formats
% information from the simulation into a plot for phase plane
% analysis
%
% Model: This simulation considers the thrusters numbered 0, 1, 6,
% & 7 as they are the ones that when coupled translate SPHERES in
% the X-axis and rotate SPHERES about its Y-axis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc; clear; close all

%% User Variables
%%%%%%%%%%%%%%%%% USER MAKES CHANGES HERE %%%%%%%%%%%%%%%%%%%%%%
% Set Initial Conditions
Pos_0         = 0;      % initial relative position of SPHERES [m]
Vel_0         = 0;      % initial relative velocity of SPHERES [m/s]
Initial       = [Pos_0; Vel_0]; % Simulation input for State Space

% Determine Basic Velocity Path
Vel_Des_Low   = .2;     % desired relative velocity (low end)  [m/s]

% Weighting Conditions
```

```
Q           = blkdiag(1,0.018);  % state weighting matrix
R           = .06;               % control weighting matrix
TAU         = 1:.1:3;            % look-ahead weight
% TAU         = 2;

% Simulation Parameters
StepSize    = 0.005;             % time step of simulation [sec]
Duration    = 45;                % length of simulation    [sec]
%%%%%%%%%%%%%%%%%% END TYPICAL USER CHANGES %%%%%%%%%%%%%%%%%%%%%

%% Constants

%% Load SPHERES Constants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mass Moments of Inertia
Eye           = [1 0 0; 0 1 0; 0 0 1];   % Identity Matrix
MOI_Wet = [ 2.30e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
             9.90e-5   2.42e-2  -2.54e-5;% SPHERES MOI w/ full tank
            -2.95e-4  -2.54e-5   2.14e-2;];
MOI_Dry = [ 2.19e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
             9.90e-5   2.31e-2  -2.54e-5;% SPHERES MOI w/empty tank
            -2.95e-4  -2.54e-5   2.13e-2;];
InMOI_Wet     = inv(MOI_Wet);            % shorthand
InMOI_Dry     = inv(MOI_Dry);            % shorthand
OneD_MOI_Wet  = MOI_Wet(2,2);            % MOI used for 1-D example
OneD_MOI_Dry  = MOI_Dry(2,2);            % MOI (dry) used for 1-D

% Mass of SPHERES
Mass_Wet   = 4.16;       % SPHERES mass with fuel in [Kg]
Mass_Dry   = 3.55;       % SPHERES mass without fuel in [Kg]
% Thruster Information
F    = 0.11;             % force of individual thruster [N]
l    = 0.193;            % length between thrusters [m]
% Preditiction Term
tau = 2;                 % determine slope of switch line (M=-1/tau)
```

```
% Define Dead Zone Range
                        % these values ensure translation stays
                        % w/in 1 cm of the desired value
High    = 0.001;        % values above this number aren't reset
Low     = 0.001;        % values below this number aren't reset


%% SPHERES Plant
A           = [0 1; 0 0];
Bu          = [0;1/Mass_Wet];
C           = eye(2);     % assumes all states provided by estimator
D           = zeros(2,1);% control does not directly affect output

%% Run Simulation
% Develop LQR
[K,¬,¬]     = lqr(A,Bu,Q,R);     % linear quadratic controller

time = 0:StepSize:Duration;
position = zeros(length(time),length(TAU));
velocity = zeros(length(time),length(TAU));
control  = zeros(length(TAU),1);
x = -1:.1:1;
S = zeros(length(x),length(TAU));

for ctr = 1:length(TAU)
    %Simulate the Closed Loop system with Non-Linearities included
    tau = TAU(ctr);
    sim('SPHERES_1D_Model_Relative_LQR_v2');
    position(:,ctr) = simout1.signals.values(:,1);
    velocity(:,ctr) = simout1.signals.values(:,2);
    control(ctr,1)  = sum(abs(simout1.signals.values(:,3)));
    %Make Switch Line
    var = -K(1)/K(2)/tau.*x;
    S(:,ctr) = var';
end
```

```
%% Format Plots
```

C.3 Script to Optimize K_d

The following code is titled 'SPHERES_Trade_Study_Quaternion.m' and is included below for reference within this thesis. This code was used to determine the derivative gain for the quaternion controller. Anyone wishing to modify this gain for different criterion should consider working from this code.

Listing C.3: Appendix3/SPHERES_Trade_Study_Quaternion_v4.m

```
%%%%%%%%%% SPHERES Quaternion PD weight Trade Study %%%%%%%%%%%%
%
% Trade Study for rotation of 3-D Model of SPHERES w/Bang-Bang
% Controller
%
% Author: Sam Barbaro          ENY-3                  20 Nov 2011
%
% Purpose: This script feeds constants & variables into the
% simulation SPHERES_3D_Model_Rotation_v2.mdl for various values
% of Kd and then records the Peak Value, Settling Time, & Control
% Usage for each ratio in order to highlight which value provides
% the most desireable results.
%
% Model: This simulation considers all thrusters number 0-11, but
% only uses the thrusters to change SPHERES orientation as opposed
% to its position in inertial space.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Clear Data and Load Constants
clc; clear; close all
% SPHERES_Constants
```

```
% global MOI_Wet MOI_Dry InMOI_Wet InMOI_Dry F l

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% USER MAKES CHANGES HERE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Input Desired Roll Pitch Yaw Data for Simulation
Theta_0   = [0;0;0];        % initial euler angles [rad]
                            % (roll,pitch,yaw)
Omega_0   = [0;0;0]';       % initial euler rates  [rad/s]
Theta_F   = [0; 0; 10*pi/180];
Step_Time = 5;              % used signal change to new Theta [sec]
% Convert Roll, Pitch, Yaw Data into Quaternions
Quat_0  = RPY2Q(Theta_0'); % [nx4] matrix
Quat_F  = RPY2Q(Theta_F'); % [nx4] matrix
Quat_0  = Quat_0';          % [4xn] matrix
Quat_F  = Quat_F';          % [4xn] matrix
% Weighting Conditions
Gain_P   = 1;               % Proportional Gain
Gain_D   = 0.31:.02:0.41; %0.33:0.01:0.37; % Derivative Gain
% Find length of the weighting condition that is changing
% len      = length(Gain_P);
len      = length(Gain_D);
% Set constant weighting conditions to same length as one the
% variable weighting condition
Kp_array = Gain_P.*ones(1,len);
Kd_array = Gain_D.*ones(1,len);
% Label which gain condition is changing
% var      = 'Proportional Gain';
var      = 'Derivative gain';
idw  = Kd_array;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% END TYPICAL USER CHANGES %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
58
   %% Constants
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   % Mass Moments of Inertia
   Eye          = [1 0 0; 0 1 0; 0 0 1];   % Identity Matrix
63 MOI_Wet = [ 2.30e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
               9.90e-5   2.42e-2  -2.54e-5;% SPHERES MOI w/ full tank
              -2.95e-4  -2.54e-5   2.14e-2;];
   MOI_Dry = [ 2.19e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
               9.90e-5   2.31e-2  -2.54e-5;% SPHERES MOI w/empty tank
68            -2.95e-4  -2.54e-5   2.13e-2;];
   InMOI_Wet    = inv(MOI_Wet);            % shorthand
   InMOI_Dry    = inv(MOI_Dry);            % shorthand
   OneD_MOI_Wet = MOI_Wet(2,2);            % MOI used for 1-D example
   OneD_MOI_Dry = MOI_Dry(2,2);            % MOI (dry) used for 1-D
73
   % Mass of SPHERES
   Mass_Wet  = 4.16;      % SPHERES mass with fuel in [Kg]
   Mass_Dry  = 3.55;      % SPHERES mass without fuel in [Kg]
   % Thruster Information
78 F   = 0.11;            % force of individual thruster [N]
   l   = 0.193;           % length between thrusters [m]
   % Define Dead Zone Range
                          % these values ensure translation stays
                          % w/in 1 cm of the desired value
83 High         = 0.002;  % values above this number aren't reset
   Low          = 0.002;  % values below this number aren't reset
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   %% Run Simulation in loop
88 % MOI  = 2e-2*eye(3,3); % override to principal axis for testing
   % IMOI = inv(MOI);
   MOI  = MOI_Wet;
   IMOI = InMOI_Wet;
```

```
%%%%%%%%%%%% Sample to Find Optimization Range %%%%%%%%%%%%%%%%%
%% Show how varying Kd affects the responses of Quaternion Error
Kd   = .3;
Kp = 1; sim('SPHERES_3D_Model_Rotation_Study_v2');
leng = length(Quaternion_Error.signals.values(:,1));
                        % find sim dimension
QE   = zeros(leng,len); T = QE; str  = cell(len,1);
                        % preset for speed
z = zeros(leng,1);      % to plot error desire
for ctr = 1:1:len
    Kp = Kp_array(ctr);
    Kd = Kd_array(ctr);
    sim('SPHERES_3D_Model_Rotation_Study_v2');
    QE(:,ctr) = Quaternion_Error.signals.values(:,1);
    T(:,ctr)  = Quaternion_Error.time(:,1);
    U = F*l*(sum(abs(Rotation_Sign.signals.values(:,1))) + ...
             sum(abs(Rotation_Sign.signals.values(:,2))) + ...
             sum(abs(Rotation_Sign.signals.values(:,3))));
    str{ctr,1} = sprintf('K_d = %2.3f   Torque = %2.3f Nm',Kd,U);
end
figure (4)
plot(T,QE,T(:,1),z,'--');
legend(str,'Desired Error'); legend('location','southeast');
ylabel('1^s^t Quaternion Error','fontsize',22);
xlabel('Time [sec]','fontsize',22)
set(gca,'fontsize',19)

% Show Response of Optimized Quaternion Controller
Kp = 1;
Kd = 0.343;
sim('SPHERES_3D_Model_Rotation_Study_v2');
%Plot Quaternion Error
%Plot Actual Quaternion Values vs. Desired Quaternion Values

% Plotting command have been removed
```

Appendix D. Code for Dead-Zone Affects

The following code is titled 'DeadZoneStudy.m' and was used to illustrate the relationship between system accuracy and fuel consumption as the bandwidth of the dead-zone non-linearity was changed. This script is included below for reference within this thesis. Anyone wishing to understand the affects of the dead-zone bandwidth on different paths should consider this studying a suitable starting point.

Listing D.1: Appendix3/DeadZoneStudy.m

```
%%%%%%%%%%%%%%%%%%%%%% SPHERES SIMULATION %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% Accuracy vs. Fuel Consumption for Dead-Zone %%%%%%%%%%%
%
% Master Script to model speed & path algorithms for MIT's SPHERES
% program
%
% Author: Sam Barbaro       AFIT ENY-3                  03 Feb 2012
%
% Purpose: This script feeds constants & variables into the
% simulation SPHERES_3D_Simulation.mdl for various paths that can
% be set by the user.  This script then plots the comparison of
% how well the states actually met the desired values.
%
% Model: This simulation considers all thrusters number 0-11, and
% is capable of changing SPHERES orientation as well as its
% position in global space.
%
% Programs Called: SPHERES_3D_Simulation_v3.mdl, skew.m,
%                  datainterp_v3.m
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc; clear; close all;

%% Plan Desired Path and Speeds for SPHERES &
```

```
%% Set Initial Conditions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% USER MAKES CHANGES HERE %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Set Range of Dead-Zone Bandwidths to Consider
Dband_Low  = 0.0001;        % Narrowest Deadzone Bandwidth
Dband_High = 0.1;           % Widest Deadzone Bandwidth
Pts        = 200;           % Number of data points to consider

% Define Path of SPHERES & Set Simulation Parameters
PATH = load('results.mat','time','Position_Ch_gf',...
    'Quat_0','velo','slpint','Initial','Point');
Duration = PATH.time(end);
StepSize = PATH.time(5) - PATH.time(4);

% Set SPHERES Initial Rates
Omega_i  = [0;0;0];         % initial euler rates          [rad/s]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% END TYPICAL USER CHANGES %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Load SPHERES Constants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mass Moments of Inertia
Eye          = [1 0 0; 0 1 0; 0 0 1];   % Identity Matrix
MOI_Wet = [ 2.30e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
            9.90e-5   2.42e-2  -2.54e-5;% SPHERES MOI w/ full tank
           -2.95e-4  -2.54e-5   2.14e-2;];
MOI_Dry = [ 2.19e-2   9.90e-5  -2.95e-4;% relative to COM [kg*m^2]
            9.90e-5   2.31e-2  -2.54e-5;% SPHERES MOI w/empty tank
           -2.95e-4  -2.54e-5   2.13e-2;];
InMOI_Wet    = inv(MOI_Wet);            % shorthand
InMOI_Dry    = inv(MOI_Dry);            % shorthand
```

```
OneD_MOI_Wet = MOI_Wet(2,2);              % MOI used for 1-D example
OneD_MOI_Dry = MOI_Dry(2,2);              % MOI (dry) used for 1-D
63
% Mass of SPHERES
Mass_Wet  = 4.16;       % SPHERES mass with fuel in [Kg]
Mass_Dry  = 3.55;       % SPHERES mass without fuel in [Kg]
% Thruster Information
68 F   = 0.11;          % force of individual thruster [N]
l   = 0.193;            % length between thrusters [m]
% SPHERES Plant
A   = [0 0 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0 0 1;...
       0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0;];
73 Bu  = [0 0 0; 0 0 0; 0 0 0; ...
      1/Mass_Wet 0 0; 0 1/Mass_Wet 0; 0 0 1/Mass_Wet];
C   = eye(6);           % assumes all states provided by estimator
D   = zeros(6,3);       % control does not directly affect output
% Path & Speed Controller Gain Information
78 Kd  = 0.352;         % Gain for Quaternion Controller
Q1  = 1;                % "Q" Weight for Position in LQR Controller
Q2  = 0.018;            % "Q" Weight for Velocity in LQR Controller
R   = 0.06;             % "R" Weight for LQR Controller
Q   = blkdiag(Q1,Q1,Q1,Q2,Q2,Q2); % "Q" Weighting Matrix
83 R   = blkdiag(R,R,R);  % "R" Weighting Matrix
[K,¬,¬] = lqr(A,Bu,Q,R);% LQR gain
% Preditiction Term
tau = 2;                % determine slope of switch line (M=-1/tau)
% Define Dead Zone Range
88 BW  = linspace(Dband_Low,Dband_High,Pts);
% Set Rate Limits
Rate_T   = .10;         % S/C will not exceed this speed    [m/s]
Rate_R   = deg2rad(6); % S/C will not spin faster than this[deg/s]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
93
%% Run Simulation
%%%%%%%%%%%%%%%%%%%% SPHERES Inspector %%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Quat_0    = PATH.Quat_0;      % initial quaternions  [dimensionless]
slpint    = PATH.slpint;
Omega_0   = deg2rad(Omega_i);% Convert to [rad/s]   (3x1)
Initial   = PATH.Initial;
    % Load Desired Data into Look-Up Tables (Global Frame)
Des_Vel_X  = PATH.velo(:,1);% Desired Velocity on X-axis
Des_Vel_Y  = PATH.velo(:,2);% Desired Velocity on Y-axis
Des_Vel_Z  = PATH.velo(:,3);% Desired Velocity on Z-axis
Targ_Pos_X = PATH.Position_Ch_gf(:,1);
Targ_Pos_Y = PATH.Position_Ch_gf(:,2);
Targ_Pos_Z = PATH.Position_Ch_gf(:,3);
Point      = PATH.Point;
    % Define Mass Moment of Inertia Matrix
MOI       = MOI_Wet;          % MMOT when SPHERES' fuel tank is full
IMOI      = InMOI_Wet;        % Inverse of MMOI of SPHERES when full
    % Run Simulation
DZBWf = [10 1 .1];
DZBWt = [1  1 1];
Fuel = zeros(length(BW),3);
Error= Fuel; E_pos = Fuel; E_vel = Fuel; E_ori = Fuel;
for CASE = 1:3
    for ctr = 1:length(BW);
        High_f =  DZBWf(CASE)*BW(ctr)/2;
        Low_f  = -DZBWf(CASE)*BW(ctr)/2;
        High_t =  DZBWt(CASE)*BW(ctr)/2;
        Low_t  = -DZBWt(CASE)*BW(ctr)/2;
        sim('Simulation');
        err   = Errors.signals.values(:,1:10);
        E_pos(ctr,CASE) = sum(sqrt(sum(err(:,1:3).^2,2)));
        E_ori(ctr,CASE) = sum(Errors.signals.values(:,11));
        Fuel(ctr,CASE)  = sum(Thrust.signals.values(:,1));
        Error(ctr,CASE) = E_pos(ctr,CASE)+E_ori(ctr,CASE);
    end
end
```

```
%% Normalize Data
Max_E   = max(Error(:,2));
Error_n = Error./Max_E;
Max_F   = max(Fuel(:,2));
Fuel_n  = Fuel./Max_F;

figure(1)
plot(BW,Fuel_n(:,1),'b',BW,Error_n(:,1),'--b',...
     BW,Fuel_n(:,2),'g',BW,Error_n(:,2),'--g',...
     BW,Fuel_n(:,3),'r',BW,Error_n(:,3),'--r');
xlabel('Dead-zone of Control Torque','Fontsize',22);
ylabel('Error & Fuel Consumption','Fontsize',22);
legend('Fuel use when DZ_f/DZ_t = 10',...
       'Error when DZ_f/DZ_t = 10',...
       'Fuel use when DZ_f/DZ_t = 1',...
       'Error when DZ_f/DZ_t = 1',...
       'Fuel use when DZ_f/DZ_t = 0.1',...
       'Error when DZ_f/DZ_t = 0.1');
set(gca,'fontsize',19);

figure(2)
plot(Fuel_n,Error_n);
xlabel('Fuel Consumption','Fontsize',22);
ylabel('Error','Fontsize',22);
legend('DZ_f/DZ_t = 10','DZ_f/DZ_t = 1','DZ_f/DZ_t = 0.1');
set(gca,'fontsize',19);
```

Bibliography

1. Miller, D., "SPHERES CAD," 2011.
2. Miller, D., Saenz-Otero, A., Wertz, J., Chen, A., Berkowski, G., Brodel, C., Carlson, S., Carpenter, D., Chen, S., Cheng, S., Feller, D., Jackson, S., Pitts, B., Perez, F., Szuminski, J., and Sell, S., "SPHERES: A Testbed For Long Duration Satellite Formation Flying In Micro-Gravity Conditions," .
3. Cobb, R., "Spacecraft Systems Engineering: Attitude Determination & Control," 2010.
4. Slotine, J.-J. E. and Li, W., *Applied Nonlinear Control*, Prentice Hall International Inc., Up-Saddle River, New Jersey, 1991.
5. Hilstad, M. O., Enright, J. P., Richards, A. G., and Mohan, S., "The SPHERES Guest Scientist Program," 2009.
6. Center, A. F. D., *Air Force Basic Doctrine*, Air University Press, Maxwell AFB, Alabama, 2003.
7. Selding, P. B. D., "Failed Telecommunications Satellite Drifts Out of Control," 23 January 2009.
8. Clark, S., "Air Force recoups costs to save stranded AEHF satellite," August 17, 2011 June 14, 2011.
9. Ray, J., "Air Force Satellite's Epic Ascent Should Finish Soon," 2011.
10. Hooper, R., "Learn About Robots: Robots in Space," 14 Jan, 2012.
11. Stansbery, E., "Orbital Debris Education Package," 2009.
12. Stansbery, E., "NASA Orbital Debris FAQs," 2009.
13. Aeronautics, N. and Administration, S., "Satellite Collision Leaves Significant Debris Clouds," 2009.
14. Oleksyn, V., "What a Mess! Experts ponder space junk problem," 2009.
15. Baucom, D. R., "Missile Defense Milestones: 1944-1997," .
16. Campbell, J. W., "Using Lasers in Space: Laser Orbital Debris Removal and Asteroid Deflection," 2000.
17. Rogers, M. E., "Lasers in Space: Technological Options for Enhancing U.S. Military Capabilities," 1997.
18. Goodman, J. L., "History of Space Shuttle Rendezvous," 2011.
19. Goodman, J. L., "History of Space Shuttle Rendezvous and Proximity Operations," *Journal of Spacecraft and Rockets*, Vol. 43, No. 5, 2006, pp. 944–958.

20. Clohessy, W. H. and Wiltshire, R. S., "Terminal Guidance System for Satellite Rendezvous," *Journal of the Aerospace Sciences*, Vol. 27, No. 9, 1960, pp. 653–659.

21. earl Park, H., Park, S.-Y., and Choi, K.-H., "Satellite Formation Reconfiguration and Station-Keeping Using State-Dependent Riccati Equation Technique," *Aerospace Science and Technology*, 2010.

22. Chien, S., Sherwood, R., Rabideau, G., Castano, R., Davies, A., Burl, M., Knight, R., Stough, T., Roden, J., Zetocha, P., Wainwright, R., Klupar, P., Gaasbeck, J. V., Cappelaere, P., and Oswald, D., "The Techsat-21 Autonomous Space Science Agent," 2002.

23. Directorate, S. V., "XSS-11 Micro Satellite Fact Sheet," 2005.

24. "DART Fact Sheet," 2006.

25. "On-Orbit Mission Updates of DARPA's Orbital Express," 2007.

26. Fejzic, A., "Development of Control and Autonomy Algorithms for Docking to Complex Tumbling Satellites," 2008.

27. Tweddle, B. E., "Computer Vision Based Navigation for Spacecraft Proximity Operations," 2010.

28. Sellers, J. J., *Understanding Space: An Introduction to Astronautics*, Mc-Graw-Hill, New York, 3rd ed., 2005.

29. Bedford, A. and Fowler, W., *Engineering Mechanics: Dynamics*, Pearson Prentice Hall, Upper Saddle River, New Jersey, 4th ed., 2005.

30. Bate, R. R., Mueller, D. D., and White, J. E., *Fundamentals of Astrodynamics*, Dover Publications, Inc., New York, 1971.

31. Strang, G., *Linear Algebra and its Applications*, Thomson, Brooks/Cole, CA, 4th ed., 2006.

32. Wie, B., *Space Vehicle Dynamics and Control Second Edition*, American Institute of Aeronautics and Astronautics, Inc., USA, 2nd ed., 2008.

33. Hall, C. D., *Spacecraft Attitude Dynamics*, 2003.

34. Sidi, M. J., *Spacecraft Dynamics & Control: A Practical Engineering Approach*, Cambridge University Press, New York, USA, 1st ed., 1997.

35. Levine, W. S.

36. Ogata, K., *Modern Control Engineering*, Prentice Hall, Boston, 5th ed., 2010.

37. Nise, N. S., *Control Systems Engineering*, John Wiley & Sons, Inc., U.S.A., 5th ed., 2008.

38. Arora, J. S., *Introduction to Optimum Design*, Elsevier Academic Press, Boston, 2nd ed., 2004.

39. Burl, J. B., *Linear Optimal Control: H2 and H[infinity] Methods*, Addison-Wesley Longman, Inc., 1999.

40. Vincent, T. L. and Grantham, W. J., *Nonlinear and Optimal Control Systems*, John Wiley & Sons, Inc., New York, 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE (DD–MM–YYYY)	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22-03-2012	Master's Thesis	October 2010 — March 2012

4. TITLE AND SUBTITLE
Satellite Relative Motion Control for MIT's SPHERES Program

5a. CONTRACT NUMBER
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Barbaro, Samuel P., Second Lieutenant, USAF

5d. PROJECT NUMBER
5e. TASK NUMBER
5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/ENY)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GA/ENY/12-M02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Professor David W. Miller
37-327
77 Massachusetts Ave
Cambridge, MA 02139

Miller, David W., PhD.
millerd@mit.edu
(617) 253-3288

10. SPONSOR/MONITOR'S ACRONYM(S)
MIT

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Autonomous formation flight concepts and algorithms have great potential to revolutionize spacecraft operations enabling missions to perform autonomous docking, in-space refueling, in-space robotic assembly, and space debris removal. Such tasks require the implementation of speed and path control algorithms to maneuver satellites along relative paths with specified rates along those paths. This thesis uses MATLAB® and SIMULINK® to design and simulate a control algorithm capable of providing relative speed and path control between satellites with a pointing error of less than two degrees, a position error of less than two millimeters, and a millimeter per second of velocity error. The enclosed research provides enhancements to Massachusetts Institute of Technology's SPHERES (Synchronized Position Hold Engage Reorient Experimental Satellites) program, a testbed for multi-object rendezvous and docking research. This control algorithm is to be used on-board the International Space Station to allow MIT's SPHERES program to continue to provide a practical intermediate step to develop, test, and validate autonomous formation spaceflight algorithms. Furthermore, the simulation tool used to develop the control algorithm allows a greater community of control engineers to interact with SPHERES purely in the MATLAB® development environment.

15. SUBJECT TERMS
Satellite, SPHERES, Relative Motion, Control, Formation, Spaceflight,

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	212	Dr. Richard Cobb, Advisor
U	U	U			19b. TELEPHONE NUMBER (Include Area Code) (937)255-3636, 4559 Richard.Cobb@afit.edu